09/987,230
2857

Patent Office
Canberra

I, LEANNE MYNOTT, MANAGER EXAMINATION SUPPORT AND SALES hereby certify that annexed is a true copy of the Provisional specification in connection with Application No. PR 1520 for a patent by CANON KABUSHIKI KAISHA filed on 16 November 2000.

WITNESS my hand this
Fifteenth day of November 2001

2

LEANNE MYNOTT
MANAGER EXAMINATION SUPPORT
AND SALES

S&F Ref: 514919

ORIGINAL

AUSTRALIA

Patents Act 1990

PROVISIONAL SPECIFICATION FOR THE INVENTION ENTITLED:

Apparatus Effecting Improved Arithmetic Coding in JPEG 2000 Entropy Coder

---

Name and Address of Applicant:

Canon Kabushiki Kaisha, incorporated in Japan, of 3-30-2, Shimomaruko, Ohta-ku, Tokyo, 146, Japan

Name of Inventor:

Yu-Ling (Linda) Chen

This invention is best described in the following statement:

APPARATUS EFFECTING IMPROVED ARITHMETIC CODING IN JPEG 2000 ENTROPY CODER

FIELD OF THE INVENTION

The present invention relates generally to the field of encoding of digital images and more particularly to JPEG 2000 entropy coding.

BACKGROUND

Since the 1980's, the Joint Photographic Experts Group (JPEG) standard for the coding of digital images has been widely used as an international standard for efficiently coding continuous-tone images. JPEG has a number of different modes including sequential, progressive, lossless, and hierarchical encoding. Numerous adaptations of this technique have been made. Nonetheless, an improved image compression system has been sought that allows greater flexibility for access to compressed data as well as for the compression of images than hitherto provided.

One such proposal has been the JPEG 2000 image coding system set out in the JPEG 2000 Committee Draft Version 1.0 and dated 9 December 1999. Amongst other things, this system provides sophisticated features allowing a single codestream to be used efficiently in numerous applications including manipulating a codestream without decoding, matching a codestream to a transmission channel, and locating and extracting data without decoding.

SUMMARY

In accordance with a first aspect of the invention, there is disclosed an apparatus for JPEG 2000 entropy coding. The apparatus includes: a context generator for generating a context for each bit of one or more coefficients in a code block; an arithmetic coder for entropy coding each bit to be coded from the code block using the context for the bit; and a FIFO coupled between the context generator and the arithmetic coder for streamlining transfer of data between the context generator and the arithmetic coder, the FIFO adapted to store each bit, the corresponding context and a repeat number of the bit and context pair.

Preferably, the context generator includes a device for generating a repeat pattern of two or more bit and context pairs in a single clock cycle. A run length repeat command may represent the repeat pattern. Still more preferably, the FIFO stores the run length repeat command as the repeat number.

The context generator provides context at variable rates.

Preferably, the arithmetic coder includes a device for accelerating coding of a codestream using the repeat pattern. The arithmetic coder may further include a device for calculating a repeat count r for two or more bits dependent upon an interval A and a current estimate of LPS probability Qe(I(CX)), where I(CX) is an index stored for a context CX. Still further, the arithmetic coder may further include: a device for entropy encoding the two or more bits in a Run Length context using one of the repeat count r and the repeat number dependent upon whether the repeat count r is greater than the repeat number.

In accordance with a second aspect of the invention, there is disclosed a method of JPEG 2000 entropy coding. The method includes the steps of: generating a context for each bit of one or more coefficients in a code block; arithmetic coding each bit to be coded from the code block using the context for the bit; and buffering using a FIFO to streamline transfer of data between the context generating step and the arithmetic coding step, the FIFO adapted to store each bit, the corresponding context and a repeat number of said bit and context pair.

In accordance with a third aspect of the invention, there is disclosed a computer program product having a computer readable medium having a computer program recorded therein for JPEG 2000 entropy coding. The computer program product includes: a computer program code module for generating a context for each bit of one or more coefficients in a code block; a computer program code module for arithmetic coding each bit to be coded from the code block using the context for the bit; and a computer program code module for providing a FIFO between the context generating and the arithmetic coding to streamline transfer of data between the context generator and the arithmetic coder, the FIFO adapted to store each bit, the corresponding context and a repeat number of the bit and context pair.

BRIEF DESCRIPTION OF THE DRAWINGS

Embodiments of the invention are described hereinafter with reference to the drawings, in which:

Fig. 1 is a block diagram of an entropy coder in accordance with the first embodiment of the invention;

Fig. 2 is a diagram illustrating an array for buffering of a code block in accordance with the embodiments of the invention;

Fig. 3 is a diagram illustrating five arrays used in the embodiments of the invention for storing information regarding a code block;

Fig. 4 is a graph plotting the number of bits coded in each pass for each bit plane;

Fig. 5 is a flow diagram illustrating the entropy encoding process with which the embodiments of the invention can be practiced;

Fig. 6 is a flow diagram illustrating the Code1Plane process for encoding;

Fig. 7 is a detailed flow diagram illustrating the Significance Propagation (SigProp) process in accordance with the embodiments of the invention;

Fig. 8 is a detailed flow diagram illustrating the magnitude refinement process in accordance with the embodiments of the invention;

Fig. 9 is a detailed flow diagram illustrating the cleanup pass process in accordance with the embodiments of the invention;

Fig. 10 is a diagram illustrating a memory structure for a code block and the corresponding code-block scan pattern;

Fig. 11 is a block diagram illustrating an entropy coding device utilising memory and a register window in accordance with another embodiment of the invention;

Fig. 12 is a diagram illustrating a scratch memory for storing significance state, coded, and magnitude refinement (MR1st) data;

Fig. 13 is a flow diagram illustrating the encoding process in accordance with an embodiment of the invention;

Fig. 14 is a flow diagram illustrating the first cleanup pass for a bypass control module;

Figs. 15A and 15B are a detailed flow diagram illustrating a special run length coding checking process in accordance with an embodiment of the invention and a related 4x4 Sigmatrix;

Fig. 16 is a schematic diagram illustrating a register buffer of the bypass control module in accordance with the embodiments of the invention;

Fig. 17 is a detailed flow diagram illustrating a significance-propagation-pass checking process in accordance with the embodiment of the invention;

Fig. 18 is a circuit level diagram illustrating a circuit to test each column for Significance Propagation bypass in accordance with the embodiment of the invention;

Fig. 19 is a schematic diagram illustrating a register buffer for the context generation module in accordance with the embodiment of the invention;

Fig. 20 is a flow diagram illustrating the processing of the context generation module in accordance with the embodiment of the invention;

Fig. 21 is a flow diagram illustrating the first cleanup pass process implemented by the context generation module in accordance with the embodiment of the invention;

Fig. 22 is a flow diagram illustrating the significance propagation pass carried out by the context generation module in accordance with the embodiment of the invention;

Fig. 23 is a flow diagram illustrating the magnification refinement process implemented by the context generation module in accordance with the embodiment of the invention;

Fig. 24 is a flow diagram illustrating a cleanup pass process for the context generation module in accordance with the embodiment of the invention;

Fig. 25 is a block diagram illustrating the structure of a FIFO in accordance with the embodiment of the invention;

Fig. 26 is a detailed flow diagram illustrating a modified arithmetic coder (CODEMPS) process in accordance with the embodiment of the invention;

Fig. 27 is a flow diagram of the modified arithmetic coder; and

Fig. 28 is a block diagram of a context label generator in accordance with the embodiment of the invention.

DETAILED DESCRIPTION

A method, an apparatus, and a computer program product are disclosed for JPEG 2000 entropy encoding. In the following description, numerous details are set forth. It will be apparent to one skilled in the art, however, that the present invention may be practised without these specific details. In other instances, well-known features are not described in detail so as not to obscure the present invention.

In the following description, components of the JPEG 2000 entropy encoding system are described as modules. A module, and in particular its functionality, can be implemented in either hardware or software. In the software sense, a module is a process, program, or portion thereof, that usually performs a particular function or related functions. In the hardware sense, a module is a functional hardware unit designed for use with other components or modules. For example, a module may be implemented using discrete electronic components, or it can form a portion of an entire electronic circuit such as an Application Specific Integrated Circuit (ASIC). Numerous other possibilities exist. Those skilled in the art will appreciate that the system can also be implemented as a combination of hardware and software modules.

The detailed description is organised as follows:

1. Overview
2. Context Generation - Register Array Based Method
3. Context Generation - Memory and Register Window Based Method
4. Context Label Generator
5. Arithmetic Coder
6. Computer Implementation.

While the specification refers to JPEG 2000 throughout, it will be apparent to those skilled in the art that the embodiments of the invention may appertain to JPEG 2000 like systems, i.e. where JPEG 2000 is not practiced exactly, but is in substance utilised with some modification.

1. Overview

In the JPEG 2000 encoding draft, a discrete wavelet transform (DWT) coefficient bits are arranged into code-blocks and coded in bitplane order using three coding passes for each bit plane. A code-block is defined as a rectangular block within a sub-band. The coefficients inside the code-block are coded a bitplane at a time, starting with the most significant bitplane having a non-zero element and ending with the least significant bit-plane.

For each bitplane in a code-block, a special code-block scan pattern is used for each of significance-propagation, magnitude refinement, and cleanup pases. Each coefficient bit is coded only once in one the three passes. The pass in which a coefficient

bit is coded depends on the conditions for that pass. For each pass, contexts are created using the significance states of 8 neighbouring coefficient bits of the coefficient bit currently being coded. The context is passed to an arithmetic coder along with the bit stream to effect entropy coding.

As the coding of each bitplane requires a three-pass scan and the context look-up from eight neighbouring coefficient bits, the embodiments of the invention provide appropriate buffering for the coefficient bits and their significance states of the coded bit and the neighbours to enable speedy coding. The buffered coefficient bits and their significance states enable the coding process to switch to the next position for coding on the fly in each pass. Thus, the delay involved in the conventional 3 pass one-by-one scan method is avoided.

The arithmetic coder is capable of coding one bit per clock cycle, even though a context generator does not deliver context at a constant speed. As a result, a FIFO is used to streamline the data between the context generator and the arithmetic coder in accordance with an embodiment of the invention as shown in Fig. 1.

Fig. 1 is a block diagram illustrating the entropy encoder 100 in accordance with an embodiment of the invention. Besides providing a general representation, Fig. 1 illustrates a first embodiment of the invention. The entropy coder 100 includes a 3-pass context generator module 110, a FIFO 120, and an arithmetic coder 130. The FIFO 120 buffers the output provided by the context generator module 110 to be provided to the arithmetic coder 130. The context generator 120, arithmetic coder 130, and the FIFO 120 are described in greater detail hereinafter.

2. Context Generation - Register Array Based Method

A buffering array 200 for a code-block is shown in Fig. 2. The array 200 has the same dimensions mxn as a code-block, having a width and a height that are constrained to the power of 2. The array 200 is further divided into scans, named scan_1 210, scan_2 212 and so on to scan_n/4 214 as per the scan pattern used in 3 passes.

As shown in Fig. 3, there are 5 arrays for the bit plane coefficients bits 310, signs of coefficients 320, significance states of coefficients 330, coded indications for bitplane coefficients 340 and 1st magnitude refinement indications 350 respectively. These 5

arrays are named Bit[n][m] 310, Sign[n][m] 320, SigState[n][m] 330, Coded[n][m] 340 and MR1st[n][m] 350 used in the coding process.

The notation ArrayName[r][c] denotes the bit located in row r and column c for the named array. Scan_1[c] means the 4 bits in column c of scan_1 (e.g. column 0 or C = 0). ArrayName[r] represents all bits in row r of the named array.

The Bit array 310 is updated with coefficient bits of a bit-plane before coding is performed on that bit-plane. The Sign array 320 is updated once only for a code-block with sign information for the coefficients. Each bit in the SigState array 330 is set when a corresponding coefficient turns significant. Each bit in the Coded array 340 is set when a corresponding bit in a bit-plane is coded in a particular pass. The bit in MR1st array 350 is set when a corresponding coefficient turns significant and is cleared after the corresponding coefficient finishes coding its first Magnitude Refinement bit.

2.1 Encoding Process

The encoding process generates a context for each one of a block of DWT transformed coefficients, which are converted to sign-magnitude format and stored in memory. The first bit-plane with a non-zero element is found using the sign-magnitude format conversion process, and the bit-plane number is then ready to be used.

Fig. 5 is a flow diagram illustrating the encoding process 500. A block of sign-magnitude format converted coefficients are stored in memory along with the number of significant bit-planes. The number of significant bit-planes in a code-block is denoted as N and indicates the number of bit-planes to be coded. The first bit-plane with a non-zero element, bit-plane(N-1), is first read by a ReadPlane (N, sign) process and then coded using a cleanup pass only. The rest of the bit-planes are then coded using a Code1Plane(N) process.

Upon the start of encoding a code-block in step 510, the SigState array 330 is reset. In step 512, the number of significant bit planes N is decremented.

In step 514, a ReadPlane (N, sign) process updates the bit-plane coefficient bits and sign bits in the Bit and Sign arrays 310, 320. The memory 1000 of Fig. 10 is structured according to the code-block particular scan-pattern. Starting at the top left, the

first 4 coefficients of the first column (rows 0, 1, 2 and 3) are stored in memory location 0. Then the first 4 coefficients of the second column are stored in the next location until the width of the code-block has been reached. Then the second 4 coefficients (4m, 4m+1, 4m+2 and 4m+3) of the first column are stored following the first 4 coefficients (4m-4, 4m-3, 4m-2, 4m-1) in the last column. Given this memory structure, each entry of scan_1 (corresponding to rows 210 of Fig. 2) for the Bit and Sign arrays 310, 320 can be updated in one clock cycle.

The buffering arrays 310-350 are capable of Rotateleft and Rotateup operations. In a Rotateleft operation, the contents in column[x] are moved to column[x-1], where x is in the range of 1 to m-1, and column[0] is moved to column[m-1] in the same clock cycle (see Fig. 2). In a Rotateup operation, the contents of a scan_y (e.g. 212) are moved to scan_y-1 (e.g. 210), where y is in the range of 2 to n/4, and scan_1 210 is moved to scan_n/4 214 in the same clock cycle.

The variation of a Rotateleft operation on the 5 arrays 310-350 is indicated in Table 1. The extra 2 rows (4 and n-1) performed on SigState and Sign array are to synchronise with rows 0 and 3 for neighbourhood context look-up. When the Sign array is first loaded, the update condition is used. Column 1 is loaded from memory when updated. Otherwise, column 1 is loaded from column 2. The sign array is loaded once for a code block. The bit array is loaded once every bit plane.

TABLE 1

Array Name	Rows Performed On	Data To Column 1	Data To Other Columns
SigState	SigState[0..4] SigState[n-1]	SigState[0..4][2] SigState[n-1][2]	SigState[0..4][0]<- SigState[0..4][1] SigState[n-1][0]<- SigState[n-1][1] SigState[0..4][m-2..2]<- SigState[0..4][m-1..3] SigState[n-1][m-2..2]<- SigState[n-1][m-1..3] SigState[0..4][m-1]<- SigState[0..4][0] SigState[n-1][m-1]<- SigState[n-1][0]
Sign	Sign[n-1]	Sign[n-1][2]	Sign[0..4][0]<- Sign[0..4][1]

Array Name	Rows Performed On	Data To Column 1	Data To Other Columns
	Sign[4] Sign[0..3]	Sign[4][2] Sign[0..3][1] from Memory (when update) Or Sign[0..3][2]	Sign[n-1][0]<- Sign[n-1][1] Sign[0..4][m-2..2]<- Sign[0..4][m-1..3] Sign[n-1][m-2..2]<- Sign[n-1][m-1..3] Sign[0..4][m-1]<- Sign[0..4][0] Sign[n-1][m-1]<- Sign[n-1][0]
Bit	Bit[0..3]	Memory (when update) Or Bit[0..3][2]	Bit[0..3][0]<- Bit[0..3][1] Bit[0..3][m-2..2]<- Bit[0..3][m-1..3] Bit[0..3][m-1]<- Bit[0..3][0]
Coded	Coded[0..3]	Coded[0..3][2]	Coded[0..3][0]<- Coded[0..3][1] Coded[0..3][m-2..2]<- Coded[0..3][m-1..3] Coded[0..3][m-1]<- Coded[0..3][0]
MR1st	MR1st [0..3]	MR1st [0..3][2]	MR1st[0..3][0]<- MR1st[0..3][1] MR1st[0..3][m-2..2]<- MR1st[0..3][m-1..3] MR1st[0..3][m-1]<- MR1st[0..3][0]

SigState [n-2, ... , n-4] are not used in coding. As a result, the RotateLeft operation is not required.

Referring back to Fig. 5, in step 516, the first Cleanup pass process is started as soon as scan_1[0] of the Bit and Sign arrays is available, which is 2 clock cycles later (one cycle to write into scan_1[1] and one cycle to shift to scan_1[0]). ReadPlane (N, sign) step 514 only writes data into scan_1[1], and the other entries in these arrays can be filled up with the mechanisms of Rotateleft and Rotateup operations. The entire Bit and Sign arrays 310, 320 are filled with the bit-plane N (already once decremented from its original value) and the corresponding sign bit of each coefficient during the course of the Cleanup pass 516. The Sign array only needs to be filled once for each code-block while the Bit array 310 needs to be filled for each bit-plane.

The SigState, Coded and MR1st arrays 330, 340, 350 are to be reset before coding starts in a code block. The Coded array 340 also needs to be reset between coding of bit planes.

Once the highest significant bit-plane is coded by the Cleanup process 516, the rest of the bit-planes are coded by a Code1Plane process. In particular, in decision block 518, a check is made to determine if N>0 (indicating bitplanes remain to be processed). If decision block 518 returns false (N), processing terminates in step 520. Otherwise, if decision block 518 returns true (Y), processing continues at step 522. In step 522, the number of significant bitplanes is decremented. In step 524, the Code1Plane (N) process is carried out (see Fig. 6). Processing then continues at decision block 518. The output of each pass is a sequence of bits and their corresponding contexts. These bit and context pair are written into a FIFO 120 for use by the arithmetic coder 130.

2.2 Code1Plane Procedure

The Code1Plane process 600 of Fig. 6 codes each bit-plane using three coding passes. Processing commences in step 610. In step 612, the ReadPlane (N, 0) process updates the Bit array 310. The sign array 320 does not need to be updated. Coding can commence as soon as scan_1[0] of Bit array 310 becomes available. Each scan of the Bit array is updated with the bit-plane bits sequentially encoding pattern scan order during one of the coding passes. In step 614 Significance Propagation (SigProp) coding is carried out. In step 616, magnitude refinement (MagRef) is performed on the bit-plane. Finally, in step 618, a CleanUp pass is performed on the bit-plane to code any remaining bits of coefficients. Processing terminates in step 620. Each of the coding passes of steps 614, 616, 618 is described in greater detail with reference to Figs. 7, 8, 9, respectively.

2.3 SigProp Procedure

Fig. 7 is a flow diagram illustrating the Significance Propagation pass 700 (corresponding to step 614 of Fig. 6). Processing commences in step 710. The number of scans to code, S, is first loaded in step 712 with nScans, which equals n/4. In decision block 714, a check is made to determine if scan_1 can be bypassed. The determination of the bypass condition is described hereinafter in section 2.6. If decision block 714 returns false (N) indicating scan_1 cannot be bypassed, processing continues at step 716 which loads column to be m. All columns in scan_1 are then coded.

In decision block 718, a check is made to determine if the column C>0. If decision block 718 returns true (Y), processing continues at step 720. In step 720, the column number C is decremented. In decision block 722, a check is made to determine if any bit in scan_1[0] is insignificant (corresponding significant state is 0) and at least one of its neighbours is significant. If decision block 722 returns false (N), processing continues at step 726. Otherwise, if decision block 722 returns true (Y), processing continues at step 724. In step 724, the bit is coded. In particular, the bit and its context label are sent to the arithmetic coder (by means of the FIFO 120). All eligible bits coded in step 724 are updated for SigState and coded arrays 330, 340. The corresponding SigState is changed if the bit turns significant and the coded bit is marked. Once all the eligible bits of column C in scan_1[0] are coded or there is no eligible bit, processing continues at step 726. In step 726, a Rotateleft operation takes place, in accordance with Table 1. The bit and sign arrays use the update condition. Processing then continues at decision block 718.

If decision block 714 returns true (Y), processing continues at step 728. Similarly, if decision block 718 returns false (N) indicating that all columns in scan_1 are coded, processing continues at step 728.

In step 728, the number of scans to code S is decremented. In step 730, a Rotateup operation is performed on all five arrays 310-350, and scan_1 becomes the next scan. In decision block 732, a check is made to determine if S>0. If decision block 732 returns false (N), processing terminates in step 734. Otherwise, if decision block 732 returns true (Y) indicating that scans remain to be coded, processing continues at decision block 714. In this manner the process continues on the next scan until all scans are coded.

2.4 MagRef Procedure

Fig. 8 is a flow diagram illustrating the magnitude refinement process 800 (corresponding to step 616 of Fig. 6). Processing commences in step 810. In step 812, the number of scans to code S is loaded with the value nScans, which equals n/4. In decision block 814, a check is made to determine if scan_1 can be bypassed. The bypass condition is described hereinafter in section 2.6. If decision block 814 returns false (N) indicating that scan_1 cannot be bypassed, processing continues at step 816. In step 816, the column number C is loaded with the value m. In decision block 818, a check is made to determine if C>0. If decision block 818 returns true (Y) indicating that further

columns remain to be processed, processing continues at step 820. In step 820, the column number C is decremented.

In decision block 822, a check is made to determine if any bit and scan_1[0] has a corresponding significant state and was not coded in the Significance Propagation pass in scan_1[0]. If decision block 822 returns false (N), processing continues at step 826. Otherwise, if decision block 822 returns true (Y), processing continues at step 824. In step 824, the bits of scan_1[0] are coded if eligible. Each of the eligible bits and its corresponding context label are sent to the arithmetic coder 130. In sep 824, the coded and MR1st arrays 340, 350 are updated accordingly. In step 826, a Rotateleft operation is performed in accordance with Table 1. For column 1 of bit array 310, the input data is from memory if the scan has not been updated; otherwise, the input is from column 2. Processing then continues at decision block 818. In this manner, the process codes through all columns in scan_1.

If decision block 814 returns true (Y), processing continues at step 828. Similarly, if decision block 818 returns false (N), processing continues at step 828. In step 828, the number of scans to code S is decremented. In step 830, a Rotateup operation is performed on all five arrays 310-350, and scan_1 becomes the next scan. In decision block 840, a check is made to determine if S>0. If decision block 840 returns false (N), processing is terminated in step 842. Otherwise, if decision block 840 returns true (Y), processing continues at step 814. In this manner, the magnitude refinement coding process continues on the next scan until all scans have been coded.

2.5 Cleanup Procedure

Cleanup pass 900 of Fig. 9 codes the remaining uncoded bits in each bit-plane after the significance propagation and magnitude refinement passes. Because there is nothing to be coded by the significance propagation and magnitude refinement passes in the most significant bit-plane, this bit plane is coded with Cleanup pass 900 only.

Fig. 9 is a flow diagram illustrating the cleanup pass process 900. Processing commences in step 910. In step 912, a check is made to determine if the whole bit plane has been coded. If decision block 912 returns true (Y), processing terminates at step 918. Otherwise, if decision block 912 returns false (N), processing continues at step 914. In step 914, the number of scans to code S is loaded with the value nScan, which is equal to

n/4. In decision block 916, a check is made to determine if S>0. If decision block 916 returns false (N), processing terminates in step 918. Otherwise, if decision block 916 returns true (Y), processing continues at step 920.

In step 920, the column number C is set equal to m and the number of scans S is decremented. In decision block 922, a check is made to determine if all bits in scan_1 have been coded. If decision block 922 returns true (Y), processing continues at step 924. In step 924, a Rotateup operation is performed to rotate to the next scan. With reference to step 924, once all columns in scan_1 have been coded, the Rotateup operation is performed on all five arrays, and scan_1 becomes the next scan. Processing then continues at decision block 916.

Otherwise, if decision block 922 returns false (N), processing continues at decision block 926. In decision block 926, a check is made to determine if C>0. If decision block 926 returns false (N), processing continues at decision block 924. Otherwise, if decision block 926 returns true (Y), processing continues at step 928.

In step 928, the column number C is decremented. In decision block 930, a check is made to determine if all bins of scan_1[0] have been coded. If decision block 930 returns true (Y), processing continues at step 938. Otherwise, if decision block 930 returns false (N), processing continues at decision block 932.

In decision block 932, a check is made to determine if all bits in scan_1[0] are not coded and have insignificant neighbours. If decision block 932 returns true (Y), processing continues in step 934. In step 934, run length coding is applied to the four bits, before processing continues at step 938. Otherwise, if decision block 932 returns false (N), processing continues at step 936. In step 936, the rest of the bits in scan_1[0] are coded.

In step 938, a Rotateleft operation is applied to scan_1, and processing then continues at decision block 926. The same process continues on the next scan until all scans have been coded.

2.6 Speed Up

As the Rotateleft and Rotateup operations each take one clock cycle to finish, these operations are overhead when there are no eligible bits in a particular pass. The overhead becomes quite significant when there are only a few bits to code in a particular pass. For example, when there is only one bit to code in the xth column of scan_3, m*2(Rotateleft for scan_1 and scan_2) + 2(Rotateup) + x(Rotateleft for scan_3) clock cycles are required to reach the bit to be coded. After this bit is coded, another m*(n/4-3) + (n/4 – 2) clock cycles are required to get the arrays ready for next pass.

Fig. 4 is a graph illustrating the average trend 410, 420, 430 of number of bits coded in each of the cleanup, Significance Propagation, and magnitude refinement passes from the most significant bit-plane to the least significant bit-plane. As the number of coded bits in a particular pass drops, the overhead for a Rotateleft operation becomes significant.

Speed up can be obtained by checking each coding pass. In the Significance Propagation pass, all bits in scan_1 are checked for if either no significant neighbours exist or the bits to be coded are all significant (see decision block 714 of Fig. 7). If the checking condition is true, a Rotateup operation can be carried out and the next scan started without coding any bits. That is, a bypass is carried out.

In the Magnitude Refinement pass, a check is made to determine if there are any bits in scan_1 which are significant and not coded (see decision block 814 of Fig. 8). If the checked-for condition is true, a Rotateup operation is carried out and next scan started without coding any bits.

In the Cleanup pass 900, a check is made to determine if all bits are coded (see decision block 912). If so, this pass is bypassed. Otherwise, the Cleanup pass process 900 checks if all bits in scan_1 are coded (see decision block 922). If so, a Rotateup operation to the next scan is carried out.

In the case of bypassing a scan in a particular pass for speed up, the update of the Bit array 310 for the current bit-plane on that scan is also omitted. The update is carried out on the next pass performed on this scan. For example, if Significance Propagation pass 700 is bypassed on scan_1, the update of Bit array 310 on scan_1 takes place in the

Magnitude Refinement pass 800 if the latter pass is not to be bypassed. Otherwise, the update takes place in the Cleanup pass 900.

There is an update status bit for each scan in the Bit array. Each scan in the Bit array has an extra update status bit (not shown in the drawings). These status bits are reset before coding on a bit-plane, and the Coded array is reset. Both the SigState and the MR1st arrays 330, 350 are reset before coding a code-block. The states of these arrays 330, 350 are maintained through the coding between bit-planes.

The scheme for bypassing a particular pass can be further refined to a smaller resolution to gain more speed. For example, a scan can be divided into a few sections and a check for the specific bypass condition for each pass can be done on each section separately. A RotateLeft operation can then be applied to Scan_1 and the whole section is bypassed for a specific pass instead of a whole scan.

2.7 Special Speed-Up For The 1st Cleanup Pass

Further speed-up can be achieved from the 1st cleanup pass in accordance with the embodiments of the invention since most of the bits in the most significant bit plane are coded with RUN LENGTH 4 context. The speed-up is accomplished with pre-collected statistics of coefficients during the sign-magnitude transform process, as described hereinafter. The statistics for a portion of a scan can be read and a decision, that if this portion can be all coded with RUN LENGTH 4 context, can be made, saving the time needed to read bit plane data of each column in this portion. The collected statistics information of coefficients in a code block are stored together with the coefficients. Before reading out the bit plane data for a portion of a code block, the statistics are read first and used to analyse if this portion needs to be coded with another context, rather than RUN LENGTH 4. If the test shows that every column in this portion can be coded with RUN LENGTH 4, a special "RUN LENGTH 4 repeat" command is sent by the context generator 110 via the FIFO 120 to the arithmetic coder 130. The context generator can then advance to the next portion.

The statistics of coefficients indicate if a portion of bits turns significant after the 1st cleanup pass. Preferably, a 32x32 code block is used with a portion covering a quarter of a scan as an example for collecting the statistics information. First, each portion is scanned through and the maximum magnitude of that portion is found and stored. As

each portion is scanned, the maximum magnitude of the code block also can be found. Table 2 shows a resulting array of maximum magnitude for each portion.

TABLE 2

3	2	4	4
6	3	2	4
4	2	2	2
2	1	2	4
2	1	2	2
2	2	2	4
2	2	2	4
2	2	2	2

$\Rightarrow$

0	0	1	1
1	0	0	1
1	0	0	0
0	0	0	1
0	0	0	0
0	0	0	1
0	0	0	1
0	0	0	0

The second (right) grid is the statistics for the code block.

The maximum magnitude of the code block (on the left hand side of Table 2) is 6. The maximum magnitude has 3 significant bit planes. Each maximum magnitude in this array is then right shifted 2 bits. The array becomes the array shown on the right hand side of Table 2. The 1 bit statistics can then be stored together with the sign magnitude transformed coefficients and used for the 1st cleanup speed up.

Before starting the 1st cleanup pass on a code block, the statistics can be read to decide if a portion can be coded entirely with RUN LENGTH 4 context. The decision is made based on the pseudo code of Table 3.

TABLE 3

```
subblockwidth = code_block_width/4;
c = current column index at the start of each subblock;
column = c/subblockwidth;
p1 = 0, p2 = 0, p3 = 0, p4 = 0;
```

```
if (scan > 0)
{

        //check the SigState on top neighbour of "X" region
        for (int i = 0; i < subblockwidth; i++)
                if (SigState[BlockHeight-1][i])
                                        p2 = 1;
        if (column < 3)
                //check the SigState on the top right neighbour of "X"
                p3 = SigState[BlockHeight-1][c+subblockwidth];
}
if (column > 0)
//check the SigState on the left neighbour of "X"
p4 = SigState[0][c-1] | SigState[1][c-1] | SigState[2][c-1] |SigState[3][c-1];
if (column > 0 && scan > 0)
        //check the SigState on the top, left neighbour of "X"
        p1 = SigState[BlockHeight-1][c-1];

//If any of the regions that have been checked is significant
if (SigMatrix[scan][column]==1 || p1==1 || p2==1 || p3==1 || p4==1)
        do normal cleanup pass;
else
send "runlength4 repeat subblockwidth" to Fifo;
```

Regions P1, P2, P3 and P4 are the top left, top, top right and left region neighbours of a region "X", respectively.

The statistics are first read into the SigMatrix array, which is the array for holding the statistics read from memory. For each portion (subblock), there are 4 neighbour subblocks to consider. Location 1 represents the neighbour subblock on the top left corner. Location 2 represents the neighbour subblock on the top. Location 3 represents the neighbour subblock on the top right corner. Location 4 represents the

neighbour subblock on the left. p1, p2, p3 and p4 represent the test results from each location, respectively.

3. Context Generation - Memory And Register Window Based Method

The register array based method described with reference to Figs. 1-10 can be implemented in a mixed structure containing memory and with register windows in accordance with a second embodiment of the invention shown in Fig. 11.

The mixed structure 1100 includes a 3 pass context generation module 1120, a FIFO 1130, and an arithmetic coder 1150, all of which correspond to the modules 110, 120, and 130 of Fig. 1, respectively. Further, the mixed structure 1100 includes a scratch memory and SigMatrix 1110, a bypass control module 1160, and coefficient memory 1140. The context generation module 1120 provides output to FIFO 1130, which in turn is coupled to arithmetic coder 1150. The 3 pass context generation module 1120 also provides an output to the scratch memory and SigMatrix module 1110. The scratch memory and Sigmatrix module 1110 provides output to the bypass control module 1160. The bypass control module 1160 has a bi-directional control bus 1162 coupled with the context generation module. The module 1160 also has a uni-directional databus 1164 coupled with the context generation module 1120 and a uni-directional databus 1166B from the uni-directional coefficient memory 1140. The coefficient memory 1140 is also uni-directionally coupled to the context generation module 1120 by uni-directional bus 1166A.

The coefficient memory module 1140 has the same structure as the coefficient memory 1000 shown in Fig. 10, in the register-array-based method. The coefficients are pre-analysed in the fashion described in Section 2.7, and the statistics are stored together along with the coefficients in the coefficient memory 1140. Instead of caching the whole SigState, Coded and MR1st bit planes in register arrays, a small register-window for each is used to cache the region being coded, while the other regions that are not being coded are kept in the scratch memory. Preferably, a region is confined to 8 columns in a scan. After the context of one of the 3 passes for a particular region is generated, the updated state of SigState, Coded and MR1st bits is written back to the scratch memory 1110. The corresponding SigMatrix bit is also updated with the result of ORing all the bits of SigState in a region, on the write back to the ScratchMem module. Each entry of the scratch memory 1110 holds the state of SigState, Coded and MR1st data for a region. A

2-port (1R/1W) memory is preferably used as the scratch memory 1110 to improve efficiency.

The context generator has 2 main modules: 3 pass context generation (CG) module 1120 and the bypass control (BC) module 1130. These two modules 1120, 1130 share the same coefficient memory (CoefMem) 1140 and scratch memory (ScratchMem) 1110. The 3-pass context generation module 1120 generates the context of a specific region for any of the three noted passes. The bypass control module 1160 looks for the location of the next region to be coded after the currently active region in any of the three passes.

The CG and BC modules 1120, 1160 work in parallel and communicate with each other. When the CG module 1120 finishes the context generation for the current region of any one of the 3 passes, the module 1120 signals the BC module 1160 that the CG module 1120 is ready to accept the next region location and data. The BC module 1160 looks for the next region for the current pass and holds the location information and data until the CG module 1120 grabs the location information and data.

Fig. 12 is a block diagram of a scratch memory 1200 for the SigState, Coded, and MR1st regions 1230, 1220, 1210, respectively. The diagram illustrates the inter-relationships of the three regions 1210, 1220, 1230. For example, the three regions of bits 1210A, 1220A, 1230A of the three array regions 1210, 1220, 1230 are stored in column 0 of the memory structure 1240. The arrays 1210, 1220, 1230 are "conceptual" arrays. The data is in fact stored in the scratch memory 1110 in the order shown in Fig. 12.

3.1 BC and CG Interface

The BC and CG modules 1120, 1160 communicate through a control bus 1162 and a data bus 1164. The BC module 1160 looks for the next region to be coded in the current pass, and also indicates which column is the first column to be coded in that region. From the CG module's perspective, Table 4 lists the signals and their functions of BC and CG interface.

TABLE 4

Signal Name	I/O	Function
Pass	I	0:1st cleanup; 1:SP; 2:MP; 3: non 1st cleanup
Bitplane	I	The to be coded bit plane number.
RL4_rdy	I	The current region can be all coded with Run Length 4 context when in the 1st cleanup pass.
Region_no	I	The next region to be coded in the current pass. Region number starts with 0 on the top left corner and increases in raster scan order. A region has 8 columns.
Column_no	I	The starting column to be coded in the next region.
Valid	I	The command from BC is valid.
Ready	O	CG is ready to accept the next command.
Data	I	SigState, Coded and MR1st

3.2 Encoding Process

Fig. 13 is a flow diagram illustrating an encoding procedure 1300 in accordance with an embodiment of the invention. Processing commences in step 1310. In step 1312, a pre-analysed coefficient summary is read into the SigMatrix 1110, and the number of significant bit planes N is decremented. The coefficient analysis in the sign magnitude process as described in Section 2.7 is utilised. The analysed summary of coefficients is read into the SigMatrix before the 1st cleanup pass. In step 1314, the 1st cleanup pass is performed.

In decision block 1316, a check is made to determine if N>0. If decision block 1316 returns false (N), processing terminates at step 1318. If decision block 1316 returns true (Y), processing continues at step 1320. In step 1320, the number of significant bit planes N is decremented. In step 1322, the Code1Plane (N) is applied to the bit plane before processing continues at decision block 1316. In this manner, the remaining bit planes are coded by the Code1Plane process.

3.3 Code1Plane Procedure

With reference to step 1322 of Fig. 13, the Code1Plane process 600 of Fig. 6 codes the remaining non-top bit planes in order. Each bit in a plane is coded in one of the

3 passes. The BC and CG modules 1160, 1120 work in parallel in each pass. The BC module 1160 is always ahead of the CG module 1120 at least by one region in each pass. The CG module 1120 may generate context in the cleanup pass for the bitplane n while the BC module 1160 is looking for the next region in the Significance Propagation pass for bitplane n-1. The following sections list the processes for both the BC and CG modules 1160, 1120 in each pass.

3.4 BC Module

3.4.1 1st Cleanup Pass

Fig. 14 is a flow diagram illustrating the 1st cleanup pass process 1400 for the BC module 1160. Processing commences in step 1408. In step 1410, the SigMatrix is read from the coefficient memory 1140. In step 1412 a check is made to determine if all of the regions have been finished or processed. If decision block 1412 returns true (Y), processing terminates in step 1414. Otherwise, if decision block 1412 returns false (N), processing continues in step 1416.

In step 1416, a check is made to determine if the current region can be coded entirely with run length for encoding and, if so, places the command accordingly on the bus. Processing then continues at decision block 1418. In decision block 1418, a check is made to determine if the CG module 1120 is ready for the command. If decision block 1418 returns false (N), processing continues at decision block 1418. Otherwise, if decision block 1418 returns true (Y), processing continues at decision block 1412.

In this manner, the BC module 1160 scans each region in roster order from the top left corner and checks if the current region can be coded with the special run length for repeat command. When the checking is done, the BC module 1160 places the command on the bus and signals the CG module 1120. The BC module 1160 then waits until the CG module 1120 grabs the command for checking the next region. Preferably, a small buffer can be placed between the BC and CG interface to prevent the CG module 1120 from holding up the BC module 1160.

The checking mechanism is similar to the one used in the register array based method, while a bit more complicated to eliminate unnecessary reads to the scratch or coefficient memory 1110, 1140, which could cause delays.

Fig. 15A is a flow diagram illustrating the run-length-coding checking process 1500. Beside certain steps of the process 1500 are shown regions that are being processed. In step 1510, processing commences. In decision block 1512 SigMatrix [X] is equal to 1. Beside decision block 1512 is shown the current region being checked that is labelled X and four neighbour regions N1, N2, N3 and N4 1540. These four regions are apart from the region being checked. Each of the neighbour regions has bits of interest in that region, which are shaded. If decision block 1512 returns true (Y), processing continues at step 1530. In step 1530, a normal cleanup pass coding is performed. Processing then terminates in step 1534.

Only those bits of interest are checked when the content of SigState array 330 is read from the scratch memory module 1110. Before proceeding with a read into the scratch memory module 1110 to update the SigState for a region, in each decision block 1516, 1520 and 1524, the corresponding bit(s) in SigMatrix is checked to avoid an unnecessary read. In decision block 1528, checking of the SigState array 330 for region N4 is read into the coefficient memory 1140 instead of the scratch memory 1110, as the neighbour region N4 is coded by the CG module 1120 and the SigState array 330 in the scratch memory 1110 is not up to date. This process of Fig. 15A is described in greater detail.

If decision block 1512 returns false (N), processing continues at decision block 1514. In decision block 1514, a check is made to determine if SigMatrix [N1] is valid and equal to 1. The SigMatrix is valid if the SigMatrix exists in the current checking location. One bit in the SigMatrix represents a 8x4 region. Each region is 8x4. There is a 4x4 SigMatrix 1570 shown in Fig. 15B corresponding to this code block. When checking region 1 for example of 1570, SigMatrix [N1][N2][N3][N4] of 1540 does not exist. When checking region 5 of 1570, SigMatrix [N1][N4] also does not exist. Thus, for a number of regions of 1570 (e.g. 1-4, 5, 9, 13), one or more neighbour regions of 1540 may not be available. If decision block 1514 returns false (N), processing continues at decision block 1518. Otherwise, if decision block 1514 returns true (Y), processing continues at step 1515. In step 1515, the ScratchMem is read. In decision block 1516, a check is made to determine if the OR operation of (bits of interest) is equal to 1. If decision block 1516 returns true (Y), processing continues at step 1530. Otherwise, if decision block 1516 returns false (N), processing continues at decision block 1518.

In decision block 1518, a check is made to determine if the SigMatrix [N2] is valid and equal to 1. Again, the SigMatrix (N2) is valid if it exists at the current checking location. Bits of interest in each region are fixed. The row 1548 of region 1546 is of interest. If decision block 1518 returns false (N), processing continues at decision block 1522. Otherwise, if decision block 1518 returns true (Y), processing continues at step 1519. In step 1519, the ScratchMem is read. In decision block 1520, a check is made to determine if OR operation of (bits of interest) is equal to 1. If decision block 1520 returns true (Y), processing continues at step 1530. Otherwise, if decision block 1520 returns false (N), processing continues at decision block 1522.

In decision block 1522, a check is made to determine if SigMatrix [N3] is valid and equal to 1. The region 1550 has a bit of interest 1552. If decision block 1522 returns false (N), processing continues at decision block 1526. Otherwise, if decision block 1522 returns true (Y), processing continues at step 1523. In step 1523, the ScratchMem is read. In decision block 1524, a check is made to determine if OR (bits of interest) is equal to 1. If decision block 1524 returns true (Y), processing continues at step 1530. Otherwise, if decision block 1524 returns false (N), processing continues at decision block 1526.

In decision block 1526, a check is made to determine if SigMatrix [N4] is valid and equal to 1. For example, the last column 1556 is of interest in region 1554. If decision block 1526 returns false (N), processing continues at step 1532. Otherwise, if decision block 1526 returns true (Y), processing continues at step 1527. In step 1527, the Coef Mem is read. In decision block 1528, a check is made to determine if OR operation of (bits of interest) is equal to 1. If decision block 1528 returns true (Y), processing continues at step 1530. Otherwise, if decision block 1528 returns false (N), processing continues at step 1532. In step 1532, run length 4 repeat coding is performed. Processing then terminates in step 1534.

3.4.2 Significance Propagation Pass

With reference to Fig. 16, the register buffers 1600 of the BC module 1160 are shown. When the BC module 1160 finishes the 1st Cleanup pass, the BC module 1160 immediately starts to look for the next region that needs to be coded in the Significance Propagation pass, if there are any remaining bitplanes. There are three register groups, which buffer the SigState 1616, Coded 1626 and MR1st 1628 regions read from the

ScratchMem module 1110 for a region being tested. For each region being tested, there are 8 neighbour regions having a SigState that needs to be checked. Neighbour regions top-left (TL) 1614, top (T) 1612, top-right (TR) 1610, left (L) 1618, bottom-left (BL) 1624, bottom (B) 1622, and bottom-right (BR) 1620 are buffered in registers, while right (R) region is read directly from the ScratchMem module 1110.

A read into the ScratchMem module 1110 to retrieve the SigState 1616 of that region is only needed when the corresponding SigMatrix bit is 1. The read starts from the 1st region having a SigMatrix bit that is "1" or a SigMatrix bit that is equal to "1" in the neighbour region. If any neighbour region does not exist, the corresponding buffer is set to zero. After a region is tested, if the next region is on the right hand side of the current region, each buffered neighbour region is shifted to the left, eg. TR to T, BR to B, T to TL and B to BL (see arrows shown in Fig. 16). If there is any bit in this region that needs to be coded in the Significance Propagation pass, the BC module 1160 signals the CG module 1120 by placing the region number and the column number on the control bus. Otherwise, the BC module 1160 moves on to the next region.

The Significance Propagation pass follows the same sequence as in 1st Cleanup pass process 1400 shown in Fig. 14. The procedure for checking whether the current region should be coded in the Significance Propagation pass is illustrated in Fig. 17.

Processing commences in step 1710. In decision block 1712, a check is made to determine if the SigState of regions L, TL and BL are all 0. That is, a check is made to determine if any of the SigStates of these regions are significant. If decision block 1712 returns false (N), processing continues at step 1720. In step 1720, column number 0 is set to be the first column to be coded. Processing then continues at step 1722. In step 1722, the right R region is read. Processing then terminates at step 1724. Otherwise, if decision block 1712 returns true (Y), processing continues at decision block 1714.

In decision block 1714, a check is made to determine if the CG module 1120 is coding the left L region using the Significance Propagation pass and the SigState is all zero currently. If decision block 1714 returns false (N), processing continues at step 1726. Otherwise, if decision block 1714 returns true (Y), processing continues at step 1716. In step 1716, the coefficient memory 1140 is read to look ahead at the SigState in the L region. That is, to make sure that the SigState in the L region does not change after

the CG module 1120 codes the L region, the BC module 1160 reads the corresponding coefficient memory 1140 to retrieve the bit and looks ahead at the possible SigState changes.

In decision block 1718, a check is made to determine if the SigState in the L region is still 0. If decision block 1718 returns false (N), processing continues at step 1720. Again this signals the CG module 1120 that column zero is the first column to be coded. Otherwise, if decision block 1718 returns true (Y), processing continues at step 1726. That is, if the SigState in the L region is still all zero, the BC module 1160 finds the first column to be coded in this region.

In step 1726, the Top and Bottom regions are read if they are not valid. The Top and Bottom regions are only valid when TOP Right and Bottom Right regions have been read from the previous region. Processing then continues at decision block 1728.

In decision block 1728, a check is made to find the first column to be coded in the current region.

The check to find the first column to be coded in this region can be implemented using a circuited 1800 to test each column for Significance Propagation bypass shown in Fig. 18. The circuit 1800 includes a NOR gate 1820, an AND gate 1822 and an OR gate 1830. The NOR gate 1820 has three inputs: a bit of interest from T region 1810, all bits of interest from the test region of the SigState 1812, and a bit of interest from the B region 1814. The output AllInsig_n produced by the NOR gate 1820 is provided as input to the OR gate 1824. The AND gate 1822 has input all the bits of interest from the test region 1812. The output of the AND gate 1822 AllSig_n is input to the OR gate 1824. The output of the OR gate 1824 is the signal Tocode_n 1830.

To find the 1st column to be coded in this region, the circuit 1800 shown in Fig. 18 is used. There are 8 groups of this circuit 1800 to test each column. The three outputs from each circuit 1800 are named AllSig_n, AllInsig_n and Tocode_n and indicate "all significant in this column", "all insignificant in this column" and "to be coded in this column", respectively. These outputs decide which column is the 1st column to be coded or if there is nothing to code in this region.

The 1st column to be coded is column_n where Tocode_n is the 1st "0" output, if AllSig_n-1 equals to "1". The 1st column to be coded is the column_n-1 where Tocode_n is the 1st "0" output, if AllSig_n-1 equals to "0".

Referring back to Fig. 17, if decision block 1728 returns true (Y), processing continues at step 1730. In step 1730, the column number to be coded is set equal to n. Processing then continues at step 1722. Otherwise, if decision block 1728 returns false (N), processing continues at step 1732. That is, if none of Tocode_n equals to "0", the BC module 1160 reads region TR and BR in step 1732.

In decision block 1734, a check is made to determine if both TR and BR are all zero. If decision block 1734 returns false (N), processing continues at step 1736. In step 1736, the column number is set equal to 7, and processing continues at step 1722. That is, in decision block 1734, the SigState of the TR and BR regions are checked. In particular, only the bits of interest shown in Fig. 16 are checked. If decision block 1734 returns true (Y) indicating that both are zero, processing continues at step 1738. In step 1738, region R is read. In decision block 1740, a check is made to determine if R="0". That is column zero is checked to determine if the SigState in column zero is all zero. If decision block 1740 returns false (N), processing continues at step 1742. In step 1742, the column number is set equal to 7. Processing then terminates in step 1724. Otherwise, if decision block 1740 returns true (Y), processing continues in step 1744. In step 1744, the BC module 1160 moves the window to the next region. Processing then continues at decision block 1712 and checking on the new region starts again.

With reference to step 1744, the next region to check is the region having a SigMatrix that is "1" or with any SigMatrix equal to "1" in the neighbour regions. If any of the TR, BR or R regions is "1", column7 is the first column to be coded, per steps 1736 and 1742, respectively.

If region R is not ready, the BC module 1160 launches a read to the ScratchMem module 1110 and signals the CG module 1120 to grab the command. Region R is loaded into the current region buffer for the BC module 1160 and the 1st column of region R is loaded into region R buffer of the CG module 1160, when the CG module 1160 grabs the command.

3.4.3 Magnitude Refinement Pass

When the BC module 1160 finishes the Significance Propagation pass 1700, the module 1160 immediately starts to look for the next region which needs to be coded in the Magnitude Refinement pass. The BC module 1160 starts from the 1st region having a SigMatrix bit that is "1". If there is any bit in this region that needs to be coded in Magnitude Refinement pass, the BC module 1160 signals the CG module 1120 by placing the region number and column number, where the 1st to be coded bit is found, on the control bus. Otherwise, the BC module 1160 moves on to the next region.

The Magnitude Refinement pass follows the same sequence as in the 1st Cleanup pass process 1400 shown in Fig. 14. Once the region is loaded into the buffer 1616 in Fig. 16, the BC module 1160 checks if there is any bit with a SigState equal to "1" while the Coded region is equal to "0". If any such bit, the BC module 1160 also loads the neighbour region having a SigMatrix bit that is equal to 1. The BC module 1160 then sets the Column_no and signals the CG module 1120. Otherwise, the BC module 1160 moves to the next region and starts checking on the new region again. The next region to check is the 1st region after the current region having a SigMatrix bit that is equal to "1".

3.4.4 Cleanup Pass

When the BC module 1160 finishes the Magnitude Refinement pass, the BC module 1160 immediately starts to look for the next region that needs to be coded in the Cleanup pass. The BC module 1160 follows the same sequence as in 1st Cleanup pass 1400 shown in Fig.14.

The BC module 1160 starts from the top left region and checks if there is any Coded bit equal to "0". If there is any such bit, the BC module 1160 also loads the neighbour region, having a SigMatrix bit that is equal to "1", to the SigState buffer. Then the BC module 1160 places the region number and column number, where the 1st to be coded bit is found, on the control bus and signals the CG module 1120.

After the CG module 1120 grabs the command, the BC module 1160 loads the next region, and starts the same checking again. The next region is the region immediately following the current region. If the next region is the region on the right hand side of the current region, each buffered neighbour region is shifted to the left, eg. TR to T, BR to B, T to TL and B to BL. If there is no bit to be coded in the current

region, the BC module 1160 then moves to the next region and starts the same checking again. When the BC module 1160 finishes checking all regions, the BC module 1160 continues the Significance Propagation checking process if there are any remaining bit planes, which have not been coded.

3.5 CG Module

The CG module 1120 has the register buffers 1900 shown in Fig. 19 for the context generation. The SigState, Coded and MR1st buffers 1910-1934, 1940, 1950 get input data from the corresponding BC module 1160, while the Sign and Bit buffers 1960, 1962 are loaded from the CoefMem module 1140 before the context is generated from each column. The buffers include TL 1910, T 1912, TR 1914, L 1920, SigState 1918, R 1916, BL 1934, B 1932, and BR 1930. The sign register 1960 contains a first one bit cell 1960A on top, a 4 bit cell 1960B to the left, another 4 bit cell 1960C, and a bottom 1 bit cell 1960D.

TL 1910, L 1920, BL 1934, and sign 1960B are in column -1. TR 1914, R, 1916, and BR 1930 are in column 8.

The SigState, Coded and MR1st data 1918, 1940, 1950 of the 1st column to be coded is loaded into column 0 of these buffers using the Column_no presented on the control bus as index and the corresponding Sign and Bit data 1960, 1962 is read from the CoefMem module 1140 using the combination of Region_no and Column_no.

In each pass, a termination circuit (not shown) is used to determine where the next column to be coded is or if the region has finished being coded. When the CG module 1120 finishes coding a region, the updated SigState, Coded and MR1st data 1918, 1940, 1950 are written back to the ScratchMem module 1160. The CG module 1120 terminates coding in a region before reaching the last column when the CG module 1120 detects nothing to code in the remaining columns. In such a case, the CG module 1120 writes back the SigState, Coded and MR1st buffer 1918, 1940, 1950 to the ScratchMem module 1110 and signals the BC module 1160 the readiness for the next region of the CG module 1120. If the region has not finished, the termination circuit sends a number of columns to rotate signal (Rotate_no) to the RotateLeft function, which rotates the next column to be coded to column 0.

Context generation happens in column 0. The 1st to be coded column is loaded into column 0 with the left-hand side neighbour column loaded into column –1 as shown in Fig. 19. The array is rotated left when the CG module 1120 finishes context generation in column 0. The next column to be coded is then located in column 0 after rotation. As a result, Sign and Bit data align with column 0.

There is a counter, Remainder, which represents the remaining columns to be coded in the current region. The termination circuit uses Remainder to determine if the region has finished coding for the current pass. As the termination is checked after coding on column 0, the Remainder counter is first loaded with (7- column_no). For example, if the first column to start coding is column 2, the Remainder counter is 5, which means columns 3-7 remain. If column 7 is the first column to code, the Remainder column is 0. When the termination circuit determines the region has not finished and sends a Rotate_no signal to the RotateLeft function, the Remainder counter is decremented by Rotate_no.

The CG module 1120 continuously waits for the command sent by the BC module 1160. When a valid command is present on the control bus, the CG module 1120 decodes the Pass signal and executes one of the 4 procedures shown in Fig. 20.

The CG flow process 2000 of Fig. 20 commences in step 2010. In decision block 2012, a check is made to determine if the bit on control bus 1162 is valid (valid = 1). If decision block 2012 returns false (N), processing continues at decision block 2012. Otherwise, if decision block 2012 returns true (Y), processing continues at decision block 2014.

In decision block 2014, a check is made to determine if the pass is equal to zero. If decision block 2014 returns true (Y), processing continues at step 2016. In step 2016, the 1st Cleanup pass is performed. Processing then continues at decision block 2012. Otherwise, if decision block 2014 returns false (N), processing continues at decision block 2018.

In decision block 2018, a check is made to determine if pass = 1. If decision block 2018 returns true (Y), processing continues at step 2020. In step 2020, the Significance Propagation pass is performed. Processing then continues at decision block

2012. Otherwise, if decision block 2018 returns false (N), processing continues at decision block 2022.

In decision block 2022, a check is made to determine if pass = 2. If decision block 2022 returns true (Y), processing continues at step 2024. In step 2024, a Magnitude Refinement pass is performed. Processing then continues at decision block 2012. Otherwise, if decision block 2022 returns false (N), processing continues at step 2026. In step 2026, a Cleanup pass is performed. Processing then continues at decision block 2012.

3.5.1 RotateLeft 1~7

The RotateLeft 1~7 operation allows columns to be skipped that have not been coded in the current pass.

When CG module 1120 finishes context generation in column 0, the module 1120 rotates the contents left by 1 to 7 columns as per the termination detection circuit output and thus keeps the column to be coded in column 0. Table 5 lists the RotateLeft operation for 3 buffers, where n is the number of columns to rotate.

CG_Coded[0..7-n]<=CG_Coded[n..7]

CG_MR1st[0..7-n]<=CG_MR1st[n..7]

CG_Coded[8-n..7]<=CG_Coded[0..n-1]

CG_MR1st[8-n..7]<=CG_MR1st[0..n-1]

Let CG_SigState be A.

TABLE 5

	n						
	1	2	3	4	5	6	7
A[-1]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]
A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]
A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[-1]
A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[-1]	A[0]
A[4]	A[5]	A[6]	A[7]	A[8]	A[-1]	A[0]	A[1]
A[5]	A[6]	A[7]	A[8]	A[-1]	A[0]	A[1]	A[2]
A[6]	A[7]	A[8]	A[-1]	A[0]	A[1]	A[2]	A[3]
A[7]	A[8]	A[-1]	A[0]	A[1]	A[2]	A[3]	A[4]
A[8]	A[-1]	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]

3.5.2 Initial Buffer Loading

The initial buffer loading depends on the Column_no signal on the control bus. For the Coded and MR1st buffers 1940, 1950, the loading is straightforward. Column_no is equal to n, and the initial loading for Coded and MR1st buffers 1940, 1950 is described in Table 6.

TABLE 6

CG_Coded[0..7-n]<=BC_Coded[n..7]
CG_MR1st[0..7-n]<=BC_MR1st[n..7]
If n > 0, the other part of the buffer is:
CG_Coded[8-n..7]<=BC_Coded[0..n-1]
CG_MR1st[8-n..7]<=BC_MR1st[0..n-1]

The initial loading for the SigState buffer 1918 is shown in Table 7. The CG_SigState is A and BC_SigState is B.

TABLE 7

CG	Source data to be loaded into CG from BC according to Column no								
	0(note 1)	0(note 2)	1	2	3	4	5	6	7
A[-1]	B[-1]	A[7-n]	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]
A[0]	B[0]	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]
A[1]	B[1]	B[1]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]
A[2]	B[2]	B[2]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	0
A[3]	B[3]	B[3]	B[4]	B[5]	B[6]	B[7]	B[8]	0	B[0]
A[4]	B[4]	B[4]	B[5]	B[6]	B[7]	B[8]	0	B[0]	B[1]
A[5]	B[5]	B[5]	B[6]	B[7]	B[8]	0	B[0]	B[1]	B[2]
A[6]	B[6]	B[6]	B[7]	B[8]	0	B[0]	B[1]	B[2]	B[3]
A[7]	B[7]	B[7]	B[8]	0	B[0]	B[1]	B[2]	B[3]	B[4]
A[8]	B[8]	B[8]	0	B[0]	B[1]	B[2]	B[3]	B[4]	B[5]

Note 1: The new region is not on the right hand side of the current region.

Note 2: The new region is on the right hand side of the current region, and n is the column where the termination happens in the current region.

3.5.3 Write Back To ScratchMem

When the CG module 1120 finishes context generation for a region, the contents of SigState, Coded and MR1st buffers 1918, 1940, 1950 are written back to the ScratchMem module 1110 to maintain the state throughout the coding of a code block. As the context generation for a region in any particular pass can be terminated before the CG module 1120 scans to the last column, the ScratchMem module 1110 is updated according to the last column where the termination happens:

The result of ORing all bits in the SigState is written to the corresponding SigMatrix bit to assist the BC module in later passes.

The column n is the column where CG finishes context generation in a region. The Coded and MR1st buffers 1940, 1950 are written back to the ScratchMem module 1110 as shown in Table 8.

TABLE 8

```
when n > 0
ScratchMem _Coded[Region_no][0..n-1]<=CG_Coded[8-n..7]
ScratchMem _MR1st[Region_no][0..n-1]<=CG_MR1st[8-n..7]
ScratchMem _Coded[Region_no][ n..7]<=BC_Coded[0..7-n]
ScratchMem _MR1st[Region_no][n..7]<=BC_MR1st[0.. 7-n]
when n = 0
ScratchMem _Coded[Region_no][0..7]<=CG_Coded[0..7]
ScratchMem _MR1st[Region_no] [0..7]<=CG_MR1st[0..7]
```

The ScratchMem update for the SigState buffer 1918 in the Region_no is shown in Table 9. The ScratchMem_SigState[Region_no] is A and the CG_SigState is B.

TABLE 9

	The last column where CG finishes							
ScratchMem	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**
A[0]	B[0]	B[-1]	B[8]	B[7]	B[6]	B[5]	B[4]	B[3]
A[1]	B[1]	B[0]	B[-1]	B[8]	B[7]	B[6]	B[5]	B[4]
A[2]	B[2]	B[1]	B[0]	B[-1]	B[8]	B[7]	B[6]	B[5]
A[3]	B[3]	B[2]	B[1]	B[0]	B[-1]	B[8]	B[7]	B[6]
A[4]	B[4]	B[3]	B[2]	B[1]	B[0]	B[-1]	B[8]	B[7]
A[5]	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]	B[-1]	B[8]
A[6]	B[6]	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]	B[-1]
A[7]	B[7]	B[6]	B[5]	B[4]	B[3]	B[2]	B[1]	B[0]

While updating the ScratchMem module, the result of ORing all the bits in the SigState array is also written to the corresponding SigMatrix bit to assist the BC module in the later passes.

3.5.4 1st Cleanup Pass

The CG module 1120 performs the 1st Cleanup pass when the Pass signal on the control bus is 0. The CG module 1120 does special Run Length 4 repeat coding when the RL4_rdy signal is 1. Fig. 21 illustrates context generation for a region when the BC module 1160 signals a 1st Cleanup pass on the control bus.

The first Cleanup pass process 2100 commences in step 2110. In step 2112, the SigState, Coded and MR1st buffers 1918, 1940 and 1950 are reset. In decision block 2114, a check is made to determine if the RL4_rdy signal is equal to 1. If decision block 2114 returns true (Y), processing continues at step 2118. In step 2118, a run length 4 repeat 8 command is sent to the FIFO. Processing then continues at step 2120. Otherwise, if decision block 2114 returns false (N), the region is coded as a normal Cleanup pass in step 2116, before proceeding to step 2120. In step 2120, the Ready signal is set to 1 and the three buffers 1918, 1940, 1950 are written back to the scratch memory module 1110. Processing then terminates in step 2122.

3.5.5 Significance Propagation Pass

The CG module 1120 performs the Significance Propagation Pass when the Pass signal on the control bus is 1. Fig. 22 illustrates a process 2200 for context generation for a region when the BC module 1160 signals a Significance Propagation pass on the control bus.

Processing commences in step 2210. In step 2212, the SigState, Coded and MR1st buffers 1918, 1940 and 1950 are loaded initially according to the column_no.

In decision block 2214, a check is made to determine if any bit in column 0 is insignificant and has significant neighbours. If decision block 2214 returns false (N), processing continues at decision block 2218. Otherwise, if decision block 2214 returns true (Y), processing continues at step 2216. In step 2216, the bits are coded. Processing then continues at step 2218. In decision block 2218, a check is made to determine if the Significance Propagation pass has finished for this region.

A termination circuit (not shown) detects if the region has finished coding or where the next column to be coded is in the current region. The circuit checks any

insignificant bit with significant neighbours in each column from column 1 to column Remainder, and terminates coding if there is not an insignificant bit with significant neighbours between column 1 to column Remainder. Thus, if decision block 2218 returns true (Y), processing continues at step 2222. In step 2222, the CG module 1120 writes the three buffers 1918, 1940, 1950 back to the scratch memory module 1110 according to the column where the Significance Propagation pass terminates and then signals the BC module 1160 for the next command. Processing then terminates in step 2224.

If decision block 2218 returns false (N), processing continues at step 2220. In step 2220, a Rotateleft operation is performed. Thus, if there is any insignificant bit with significant neighbours between column 1 to column Remainder, the termination circuit sends Rotate_no signal to RotateLeft function and the next column to be coded is rotated to column 0 for coding. Rotate_no is the column where the next column to be coded is found. For example if the next column to be coded is found in column 3, Rotate_no is 3. Remainder is decremented by Rotate_no after RotateLeft operation. When Remainder is 0, the termination circuit does not perform any checking but terminates the coding. Processing then continues at decision block 2214.

3.5.6 Magnitude Refinement Pass

The CG module 1120 performs the Magnitude Refinement Pass when the Pass signal on the control bus is 2. Fig. 23 is a flow diagram illustrating a process 2300 for context generation for a region when the BC module 1160 signals Magnitude Refinement pass on the control bus. Processing commences in step 2310. In step 2312, the SigState, Coded and MR1st buffers 1918, 1940, 1950 are loaded initially according to the column number column_no.

In decision block 2314, a check is made to determine if any bit in column 0 is significant and not coded. If decision block 2314 returns false (N), processing continues at step 2318. Otherwise, if decision block 2314 returns true (Y), processing continues at step 2316. In step 2316 the bits are coded. Processing then continues at decision block 2318. In decision block 2318, a check is made to determine if the Magnitude Refinement pass has finished for the current region.

A termination circuit (not shown) detects if the region has finished being coded or if the next column to be coded is in the current region. The circuit checks any significant and not coded bit in each column from column 1 to column Remainder, and terminates coding if there is not any significant and not coded bit from column 1 to column Remainder. If decision block 2318 returns true (Y), processing continues at step 2322. Step 2322 writes the 3 buffers back to the ScratchMem module 1110 according to the column where the Magnitude Refinement pass terminates and signals the BC module 1160 for next command. Processing terminates in step 2324.

If decision block 2318 returns false (N), processing continues at decision block 2320. In step 2320, a RotateLeft operation is performed upon the region. Thus, if there is any significant and not coded bit between column 1 to column Remainder, the termination circuit sends Rotate_no signal to RotateLeft function and the next column to be coded is rotated to column 0 for coding. Rotate_no is the column where the next column to be coded is found. For example if the next column to be coded is found in column 3, Rotate_no is 3. Remainder is decremented by Rotate_no after RotateLeft operation. When Remainder is 0, the termination circuit does not perform any checking but terminates the coding. Processing then continues at decision block 2314.

3.5.7 Cleanup Pass

The CG module 1120 performs the Cleanup Pass when the Pass signal on the control bus is 3. Fig. 24 illustrates the Cleanup process 2400 of the CG module 1120 for context generation for a region when the BC module 1160 signals a Cleanup pass on the control bus. Processing commences in step 2410. In step 2412, the SigState, Coded, and MR1st buffers 1918, 1940, 1950 are initially loaded according to the column number.

In decision block 2414, a check is made to determine if any bit in column 0 for the region is not coded. If decision block 2414 returns false (N), processing continues at decision block 2418. Otherwise, if decision block 2414 returns true (Y), processing continues at step 2416. In step 2416, the bits are coded. Processing then continues at decision block 2418. In decision block 2418, a check is made to determine if the whole region is finished coding. With reference to decision block 2418, the check is made by simply ANDing the bits of the Coded buffer. A "1" indicates the whole region is coded.

A termination circuit (not shown) detects if the region has finished coding or if the next column to be coded is in the current region. The circuit checks any not coded bit in each column from column 1 to column Remainder and terminates coding if all bits are all coded from column 1 to column Remainder. If decision block 2418 returns true (Y), processing continues at step 2422. In step 2422, the CG module 1120 resets the coded buffer 1940 and writes the 3 buffers 1918, 1940, 1950 back to the scratch memory module 1110 according to the column where the Cleanup pass terminates. The CG module 1120 then signals the BC module 1160 for the next command by setting ready equal to 1. Processing then terminates in step 2424.

If decision block 2418 returns false (N), processing continues at step 2420. In step 2420, a RotateLeft operation is performed upon the region, and processing then continues at decision block 2414. If there is any not coded bit between column 1 to column Remainder, the termination sends Rotate_no signal to RotateLeft function and the next column to be coded is rotated to column 0 for coding. Rotate_no is the column where the next column to be coded is found. For example if the next column to be coded is found in column 3, Rotate_no is 3. Remainder is decremented by Rotate_no after RotateLeft operation. When Remainder is 0, the termination circuit does not perform any checking but terminates the coding.

4. Context Label Generator

The context label for each pass can be easily generated by a context label generator 2800 as shown in Fig. 28 according to the context defined in the JPEG 2000 standard draft when the required bits, eg. significance state for the neighbours, are made available. There are two kinds of context generated by the "data context label encoder" 2818 and the "sign context label encoder" 2822 respectively. The appropriate MR1st bit, significance state neighbours and sign neighbours are selected according to the bit position in column 0. These values are input to multiplexers 2810, 2812, and 2814, respectively. The bit position (0, 1, 2, 3) is used to control operation of the multiplexers 2810, 2812, 2814. Multiplexer 2810 provides 1stMR for the coefficient as input to a data context label encoder 2818. Subband and pass are inputs also to encoder 2818.

The data context label encoder 2818 generates context labels according to the current pass and subband of the code block, the pass signal is used to select between

Table 10 and 13 listed hereinafter, and the subband signal is used to select the encoding condition to be used in Table 10 for the encoder 2818.

The output of the multiplexer 2812 is coupled to an hvd counter 2816 and hv contribution encoder 2820. The output of hvd encoder 2816 is a further input to the encoder 2818.

The multiplexer 2814 provides an input to the hv contribution encoder 2820. In turn, encoder 2820 provides an input to a sign context label encoder 2822. Both encoders 2818, 2822 provide inputs another multiplexer 2824, which has sign/data as control inputs and outputs the context label. The selected significance state neighbours are fed into hvd counter 2816, which counts the number of significance bits in the horizontal, vertical and diagonal neighbours, respectively. The output of the hvd counter 2816 (h count, v count and d count) and selected MR1st bit are fed into the data context label encoder 2818, which encodes the context label according to the current pass and subband. When coding the significance propagation pass and the cleanup pass, Table 10 describes the encoding of the context label. When coding the magnitude refinement pass, Table 13 is used to encode the context label.

The selected significance state neighbours from multiplexer 2812 are also fed into the hv contribution encoder 2820 along with the selected sign neighbours using the contribution method defined in Table 11. The output of horizontal and vertical contribution is then fed into the sign context label encoder 2822 using the method defined in Table 12. The final context label output by multiplexer 2824 is selected depending on if the current bit is a data or sign bit, which is input to control the multiplexer 2824.

This context label generator works with both the register array based method and the register window and memory based method.

TABLE 10

LL and LH sub-bands	HL sub-band	HH sub-band	*Context label*[a]

ΣH	ΣV	ΣD	ΣH	ΣV	ΣD	Σ(H+V)	ΣD	
2	X[b]	X	X	2	X	X	≥3	8
1	≥1	X	≥1	1	X	≥1	2	7
1	0	≥1	0	1	≥1	0	2	6
1	0	0	0	1	0	≥2	1	5
0	2	X	2	0	X	1	1	4
0	1	X	1	0	X	0	1	3
0	0	≥2	0	0	≥2	≥2	0	2
0	0	1	0	0	1	1	0	1
0	0	0	0	0	0	0	0	0

a. Note that the context labels are numbered only for identification. The actual identifiers used is a matter of implementation.

b. X = do not care.

TABLE 11

$V_0(orH_0)$	$V_1(orH_1)$	V(or H) contribution
Significant, positive	Significant, positive	1
Significant, negative	Significant, positive	0
Insignificant	Significant, positive	1
Significant, positive	Significant, negative	0
Significant, negative	Significant, negative	-1
Insignificant	Significant, negative	-1
Significant, positive	Insignificant	1
Significant, negative	Insignificant	-1
Insignificant	Insignificant	0

TABLE 12

Horizontal contribution	Vertical contribution	XORbit	Context label
1	1	0	13
1	0	0	12
1	-1	0	11
0	1	0	10
0	0	0	9
0	-1	1	10
-1	1	1	11
-1	0	1	12
-1	-1	1	13

TABLE 13

$\sum H+\sum V+\sum D$	1st refinement for this coefficient	Context label
X[a]	False	16
≥1	True	15
0	True	14

a. "X" indicates a "don't care" state.

5. Arithmetic Coder

An arithmetic coder codes 1 bit per clock cycle as the probability is adaptively estimated when the new bit and its context are received. A sequence of bits to be coded and their context may be known. In such a case, the coder according to the embodiments of the invention can generate a new code word for the sequence of bits as the probability is known.

Thus, in accordance with the embodiments of the invention, apparatuses for JPEG 2000 entropy coding 100 (1100) include a context generator 110 (1120, 1110, 1160, 1140), an arithmetic coder 130 (1150), and a FIFO 120 (1130). A detailed diagram

of a FIFO 2500 in accordance with the embodiment of the invention is shown in Fig. 25, as described hereinafter. The context generator 110 (1110, 1120, 1140, 1160) generates a context (CX) for a bit of a coefficient in a code block. The arithmetic coder 130 (1150) entropy codes the bit of the code block. The FIFO 120 (1130) is coupled between the context generator 110 (1110, 1120, 1140, 1160) and the arithmetic coder 130 (1150) and stores the bit, the context and the number of the bit and context pair. The context generator 110 (1110, 1120, 1140, 1160) is able to generate a repeat pattern (Repeat) of two or more bit and context pairs in a single clock cycle.

In the 1st Cleanup pass, most of the bits are coded with a RUN LENGTH 4 context. The collected statistics are first read before reading any bitplane data. Once a special 1st cleanup speed up condition is met, the RUN LENGTH 4 context and the number of repeats are sent to the FIFO 2500. The FIFO 2500 contains an entry 2510 for 1 bit of data, an entry 2520 for 5 bits of context (CX), and an entry 2530 for 4 bits of repeat (Repeat). A zero repeat is preferably interpreted as 16. In accordance with Fig. 1, this FIFO 2500 can be implemented as FIFO 120 of Fig. 1 between the context generator 110 and the arithmetic coder 130. The arithmetic coder 130 of Fig. 1 then processes the "repeat" command. The time required to process this command depends on the state of the arithmetic coder 130.

Fig. 27 shows the overall processing 2700 of the arithmetic coder. Processing commences in step 2710. In decision block 2712, a check is made to determine if the FIFO 2500 is empty. If decision block 2712 returns true (Y), processing continues at decision block 2712. Otherwise, if decision block 2712 returns false (N), processing continues in step 2714.

In step 2714, the arithmetic coder fetches a set of Bit, CX and Repeat from the FIFO, which is not empty. In decision block 2716, a check is made to determine if Bit is the MPS with the context CX given. If decision block 2716 returns true (Y), CodeMPS process is executed in step 2718. Otherwise, if decision block 2716 returns false (N), CodeLPS process is executed in step 2720. From either step 2718 oe 2720, processing continues at step 2727. After finishing coding a set of Bit, CX and Repeat, a check is made in decision block 2722 to determine if the Finish condition is met. The Finish condition is a combination of the FIFO being empty and the indication of completion of

context generation from the context generator for a pass or a whole code block. The entropy coder can choose to terminate after a pass or a whole code block. If the Finish condition is not met (N), processing continues at decision block 2712 and the next set of data is fetched from the FIFO. Otherwise, if decision block 2722 returns true (Y), step 2724 flushes the arithmetic coder. Processing terminates in step 2726.

The speed up can take place when MPS is 0 and the upcoming command has a repeat number greater than 1 and its context is RUN LENGTH.

At least portions of the apparatuses 100 (1100) can be implemented as a computer program product. That is, aspects of the embodiment may be implemented in software. The computer program product includes a computer readable medium with a computer program for JPEG 2000 entropy coding recorded on the medium. The computer program product may also contain relevant data and data structures. The functionality employed by the apparatus can be implemented as code modules of the computer program. The functionality of this embodiment of the invention is described in greater detail hereinafter.

The apparatuses 100 (1100) pre-analyze transform coefficients of a code block in sign-magnitude form. Bit data 310, sign data 320, significance state data 330, coded data 340, and magnitude refinement data 350 for the code block are buffered using the register arrays for context generation. Statistical data about the coefficients is then stored, preferably with the the coefficients in the coefficient memory 1000. Using this statistical data, one or more repeat commands described hereinbefore are generated for one or more sequences of bit and context pairs for arithmetic encoding by the coder 130 (1150). In the embodiment of the invention, a specific region of the code block can be switched to in any one of significance propagation, magnitude refinement, and cleanup coding passes using rotate-left and rotate-up operations of the register arrays. A region of a code block currently being coded is buffered using a register window and the remaining regions of the code block are separately buffered using the scratch memory 1110.

The bypass control module 1160 looks ahead for the next region of a code block to be coded in each of significance propagation, magnitude refinement, and cleanup coding passes. The context generation module1120 generates the context of a region

previously provided by the bypass control module 1160. Preferably, the bypass control and context generations modules 1120, 1160 operate in parallel.

The arithmetic coder 1150 in accordance with the embodiment of the invention implements a modified CODEMPS process 2800 shown in Fig. 28, corresponding to step 2718 of Fig. 27. The circled special path 2840 in the diagram highlights the modification. Steps 2812, 2814, and 2832 are also modifications. Steps 2816 –2830 form the standard CODEMPS process described in Appendix B of the draft for JPEG 2000 Image Coding System, JPEG 2000 Committee Draft Version 1.0, dated 9 December 1999.

For the coder, CX is a label that represents the context used to produce compressed data. CX selects the probability estimate to use during the encoding of data or decision D. The binary arithmetic coding is based on the recursive probability interval subdivision of Elias coding. With each binary decision the current probability interval is subdivided into two sub-intervals. The code string may be modified to point to the base or lower bound of the probability sub-interval assigned to the symbol. For the partitioned current interval, a sub-interval the more probable symbol (MPS) is ordered above the sub-interval for the less probable symbol (LPS).

Coding operations are performed using fixed precision integer arithmetic and use an integer representation of fractional values. In particular, decimal 0.75 is represented by 0x8000. An interval A is kept in the range of 0.75 <= A < 1.5 (where "<=" means less than or equal to) by doubling the interval whenever the integer value is lower than 0x8000. C represents the code register. Qe is the current estimate of the LPS probability. A-Qe is the sub-interval of the MPS, and Qe is the sub-interval for the LPS. D and CX are read and passed for encoding as pairs, and this continues until all pairs have been read. In Fig. 26, RENORME stands for Renormalization in the encoder. Further information about the standard aspects of CODEMPS encoding can be found in Appendix B of the draft for JPEG 2000 Image Coding System, JPEG 2000 Committee Draft Version 1.0, dated 9 December 1999.

I(CX) is an index stored for the context CX. NMPS represents the next MPS index.

Referring to Fig. 26, processing commences in step 2610. In step 2612, the number of repeats "r" is calculated at the floor value of (A-0x8000)/Qe(I(CX)). That is, the number of repeats r that can be processed before A is smaller than 0x8000 at which the probability is updated is first calculated. In decision block 2614, a check is made to determine if r >1. From an AC point of view, the speed-up is not restricted to RUN LENGTH only. However from the context generator point of view RUN LENGTH is the only occasion in which Repeat can be greater than 1. If decision block 2614 returns true (Y), processing continues via the modification beginning at step 2642.

Regarding the modifications, in decision block 2642, the number of repeats (r) that can be processed is compared with the repeat number (Repeat) in the command to determine if r is greater than Repeat. If decision block 2642 returns true (Y) indicating that r is greater than Repeat, processing continues at step 2646. That is, the command can be processed in a clock cycle. In step 2646, the following actions are performed: (1) A = A-Repeat*Qe(I(CX)); (2) C=C+Repeat*Qe(I(CX)); and Repeat = 0. That is, the command updates the interval and the code word register, and sets Repeat equal to zero. Processing continues at decision step 2633. In step 2633, a check is made to determine if Repeat > 0. If decision block 2633 returns true (Yes), processing continues at step 2612. Otherwise, if step 2633 returns false (No), processing terminates in step 2634.

Otherwise, if decision block 2642 returns false (N) indicating that r < Repeat, processing continues at step 2644. In step 2644, the following actions are performed: (1) A=A-r*Qe(I(CX)); C=C+ r*Qe(I(CX)); and (3) Repeat=Repeat-r. That is, the command updates the interval and the code word register, and reduces Repeat by r. The same command with the reduced Repeat is processed in the next clock cycle. Processing continues at decision step 2633.

Otherwise, if decision block 2614 returns false (N) indicating that the number of repeats is not greater than 1 or the context is not RUN LENGTH, a normal coding path is taken (steps 2616 – 2630). In step 2616, the interval A is updated (A=A-Qe(I(CX))). In decision block 2618, a check is made to determine if ANDing A and 0x8000 is equal to zero. This indicates that A is too small (<0.75). If decision block 2618 returns false (N), processing continues at step 2620. In step 2620, the code word register is updated (C=C+Qe(I(CX))). Processing continues at step 2632 and Repeat is decremented by one. Processing then continues at decision step 2633. Otherwise, if decision block 2618

returns true (Y), processing continues at decision block 2622. In decision block 2622, a check is made to determine if A < Qe(I(CX)). If decision block 2622 returns false (N), processing continues at step 2626. In step 2626, the code word register is updated as per step 2620. Processing then continues at step 2628. Otherwise, if decision block 2622 returns true (Y), processing continues at step 2624. In step 2624, the interval is set equal to Qe(I(CX)). Processing then continues at step 2628.

In step 2628, the index I(CX) is set equal to the next index NMPS(I(CX)). In step 2630, renormalization in the encoder is performed. Processing then continues in step 2632 where Repeat is decremented by one, before continuing at decision step 2633.

6. Computer Implementation

Aspects of the embodiments of the invention can implemented using a general-purpose computer. In particular, the processing or functionality can be implemented as software, or a computer program, executing on the computer. The method or process steps for JPEG 2000 entropy encoding may be effected by instructions in the software that are carried out by the computer. The software may be implemented as one or more modules for implementing the process steps. A module is a part of a computer program that usually performs a particular function or related functions. Also, as described hereinbefore, a module can also be a packaged functional hardware unit for use with other components or modules.

The software may be stored in a computer readable medium, including the storage devices described below. The software is preferably loaded into the computer from the computer readable medium and then carried out by the computer. A computer program product includes a computer readable medium having such software or a computer program recorded on it that can be carried out by a computer. The use of the computer program product in the computer preferably effects advantageous apparatuses for trading goods or services or both using an electronic network.

The computer readable medium can include one or more of the following: a floppy disc, a hard disc drive, a magneto-optical disc drive, CD-ROM, magnetic tape or any other of a number of non-volatile storage devices well known to those skilled in the art. The program may be supplied to the user encoded on a CD-ROM or a floppy disk, or alternatively could be read by the user from the network via a modem device connected to

the computer, for example. Still further, the software can also be loaded into the computer system from other computer readable medium including magnetic tape, a ROM or integrated circuit, a magneto-optical disk, a radio or infra-red transmission channel between the computer and another device, a computer readable card such as a PCMCIA card, and the Internet and Intranets including email transmissions and information recorded on websites and the like. The foregoing are merely examples of relevant computer readable mediums. Other computer readable mediums may be practiced without departing from the scope and spirit of the invention.

The foregoing is simply provided for illustrative purposes and other configurations can be employed without departing from the scope and spirit of the invention. Computers with which the embodiment can be practiced include IBM-PC/ATs or compatibles, one of the Macintosh (TM) family of PCs, Sun Sparcstation (TM), a workstation or the like. The foregoing are merely examples of the types of computers with which the embodiments of the invention may be practiced. Typically, the processes of the embodiments, are resident as software or a program recorded on a hard disk drive as the computer readable medium, and read and controlled using the processor. Intermediate storage of the program and intermediate data and any data fetched from the network may be accomplished using the semiconductor memory, possibly in concert with the hard disk drive.

In the foregoing manner, a method, an apparatus, and a computer program product for JPEG 2000 entropy encoding are disclosed. While only a small number of embodiments are described, it will be apparent to those skilled in the art in view of this disclosure that numerous changes and/or modifications can be made without departing from the scope and spirit of the invention.

APPENDIX A

This appendix contains Annex C "Arithmetic Entropy Coding" of JPEG 2000 Image Coding Ssytem, JPEG 2000 Part I Final Draft International Standard, dated 18 August 2000 (ISO/IEC FDIS 15444-1 2000; ITU-T REC T.800).

Annex C

Arithmetic entropy coding

(This Annex forms a normative and integral part of this Recommendation I International Standard.)

In this Annex and all of its subclauses, the flow charts and tables are normative only in the sense that they are defining an output that alternative implementations shall duplicate.

C.1 Binary encoding (informative)

Figure C-1 shows a simple block diagram of the binary adaptive arithmetic encoder. The decision (D) and context (CX) pairs are processed together to produce compressed image data (CD) output. Both D and CX are provided by the model unit (not shown). CX selects the probability estimate to use during the coding of D. In this International Standard, CX is a label for a context.

C.1.1 Recursive interval subdivision (informative)

The recursive probability interval subdivision of Elias coding is the basis for the binary arithmetic coding process. With each binary decision the current probability interval is subdivided into two sub-intervals, and the code string is modified (if necessary) so that it points to the base (the lower bound) of the probability sub-interval assigned to the symbol which occurred.

In the partitioning of the current interval into two sub-intervals, the sub-interval for the more probable symbol (MPS) is ordered above the sub-interval for the less probable symbol (LPS). Therefore, when the MPS is coded, the LPS sub-interval is added to the code string. This coding convention requires that symbols be recognized as either MPS or LPS, rather than 0 or 1. Consequently, the size of the LPS interval and the sense of the MPS for each decision must be known in order to code that decision.

Since the code string always points to the base of the current interval, the decoding process is a matter of determining, for each decision, which sub-interval is pointed to by the compressed image data. This is also done recursively, using the same interval sub-division process as in the encoder. Each time a decision is decoded, the decoder subtracts any interval the encoder added to the code string. Therefore, the code string in the decoder is a pointer into the current interval relative to the base of the current interval. Since the coding process involves addition of binary fractions rather than concatenation of integer code words, the more probable binary decisions can often be coded at a cost of much less than one bit per decision.

C.1.2 Coding conventions and approximations (informative)

The coding operations are done using fixed precision integer arithmetic and using an integer representation of fractional values in which 0x8000 is equivalent to decimal 0,75. The interval A is kept in the range $0{,}75 \leq A < 1{,}5$ by doubling it whenever the integer value falls below 0x8000.

The code register C is also doubled each time A is doubled. Periodically – to keep C from overflowing – a byte of compressed image data is removed from the high order bits of the C-register and placed in an external compressed image data buffer. Carry-over into the external buffer is prevented by a bit stuffing procedure.

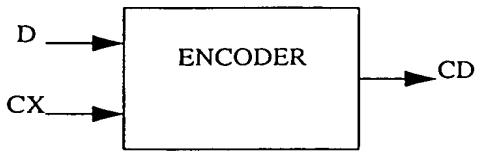

Figure C-1 — Arithmetic encoder inputs and outputs

Keeping A in the range $0,75 \leq A < 1,5$ allows a simple arithmetic approximation to be used in the interval subdivision. The interval is A and the current estimate of the LPS probability is Qe, a precise calculation of the sub-intervals would require:

$$A - (Qe * A) = \text{sub-interval for the MPS} \quad \text{C.1}$$

$$Qe * A = \text{sub-interval for the LPS} \quad \text{C.2}$$

Because the value of A is of order unity, these are approximated by

$$A - Qe = \text{sub-interval for the MPS} \quad \text{C.3}$$

$$Qe = \text{sub-interval for the LPS} \quad \text{C.4}$$

Whenever the MPS is coded, the value of Qe is added to the code register and the interval is reduced to A - Qe. Whenever the LPS is coded, the code register is left unchanged and the interval is reduced to Qe. The precision range required for A is then restored, if necessary, by renormalization of both A and C.

With the process illustrated above, the approximations in the interval subdivision process can sometimes make the LPS sub-interval larger than the MPS sub-interval. If, for example, the value of Qe is 0,5 and A is at the minimum allowed value of 0,75, the approximate scaling gives 1/3 of the interval to the MPS and 2/3 to the LPS. To avoid this size inversion, the MPS and LPS intervals are exchanged whenever the LPS interval is larger than the MPS interval. This MPS/LPS conditional exchange can only occur when a renormalization is needed.

Whenever a renormalization occurs, a probability estimation process is invoked which determines a new probability estimate for the context currently being coded. No explicit symbol counts are needed for the estimation. The relative probabilities of renormalization after coding an LPS or MPS provide an approximate symbol counting mechanism which is used to directly estimate the probabilities.

C.2 Description of the arithmetic encoder (informative)

The ENCODER (Figure C-2) initializes the encoder through the INITENC procedure. CX and D pairs are read and passed on to ENCODE until all pairs have been read. The probability estimation procedures which provide adaptive estimates of the probability for each context are imbedded in ENCODE. Bytes of compressed image data are output when necessary. When all of the CX and D pairs have been read, FLUSH sets the contents of the C-register to as many 1 bits as possible and then outputs the final bytes. FLUSH also terminates the encoding and generates the required terminating marker.

NOTE — While FLUSH is required in ITU-T Rec.T.88 | ISO/IEC 14492, it is informative in this specification. Other methods, such as that defined in Annex D.4.2, are acceptable.

C.2.1 Encoder code register conventions (informative)

The flow charts given in this Annex assume the register structures for the encoder shown in Table C-1.

Table C-1 — Encoder register structures

	MSB			LSB
C-register	0000 cbbb	bbbb bsss	xxxx xxxx	xxxx xxxx
A-register	0000 0000	0000 0000	1aaa aaaa	aaaa aaaa

The "a" bits are the fractional bits in the A-register (the current interval value) and the "x" bits are the fractional bits in the code register. The "s" bits are spacer bits which provide useful constraints on carry-over, and the "b" bits indicate the bit positions from which the completed bytes of the compressed image data are removed from the C-register. The "c" bit is a carry bit.

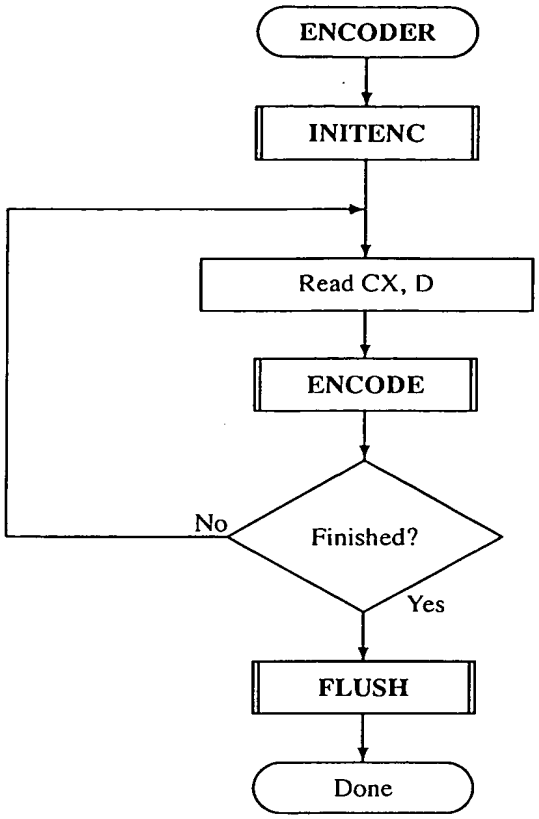

Figure C-2 — Encoder for the MQ-coder

The detailed description of bit stuffing and the handling of carry-over will be given in a later part of this Annex.

C.2.2 Encoding a decision (ENCODE) (informative)

The ENCODE procedure determines whether the decision D is a 0 or not. Then a CODE0 or a CODE1 procedure is called appropriately. Often embodiments will not have an ENCODE procedure, but will call the CODE0 or CODE1 procedures directly to code a 0-decision or a 1-decision. Figure C-3 shows this procedure.

C.2.3 Encoding a 1 or a 0 (CODE1 and CODE0) (informative)

When a given binary decision is coded, one of two possibilities occurs – the symbol is either the more probable symbol or it is the less probable symbol. CODE1 and CODE0 are illustrated in Figure C-4 and Figure C-5. In these figures, CX is the context. For each context, the index of the probability estimate which is to be used in the coding operations and the MPS value are stored. MPS(CX) is the sense (0 or 1) of the MPS for context CX.

C.2.4 Encoding an MPS or LPS (CODEMPS and CODELPS)

The CODELPS (Figure C-6) procedure usually consists of a scaling of the interval to Qe(I(CX)), the probability estimate of the LPS determined from the index I stored for context CX. The upper interval is first calculated so it can be compared to the lower interval to confirm that Qe has the smaller size. It is always followed by a renormalization (RENORME). In the event that the interval sizes are inverted, however, the conditional MPS/LPS exchange occurs and the upper interval is coded. In either case, the probability estimate is updated. If the SWITCH flag for the index I(CX) is set, then the MPS(CX) is inverted. A new index I is saved at CX as determined from the next LPS index (NLPS) column in Table C-2.

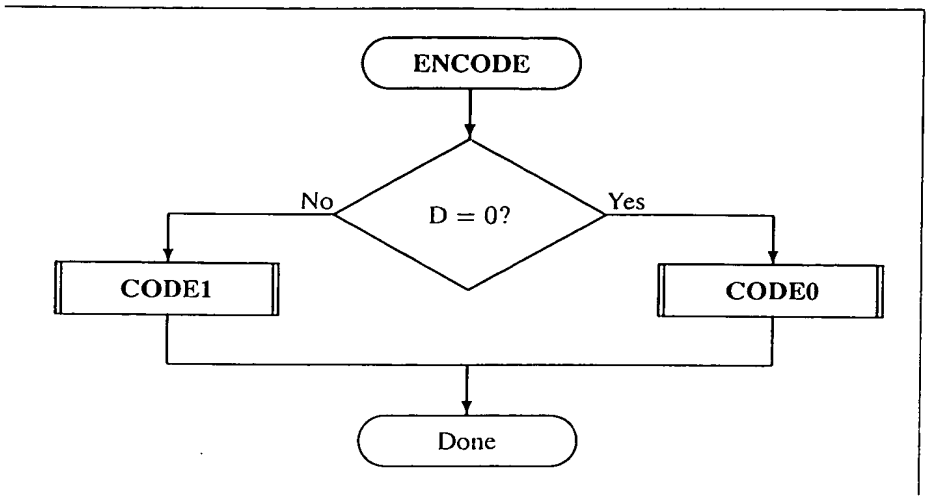

Figure C-3 — ENCODE procedure

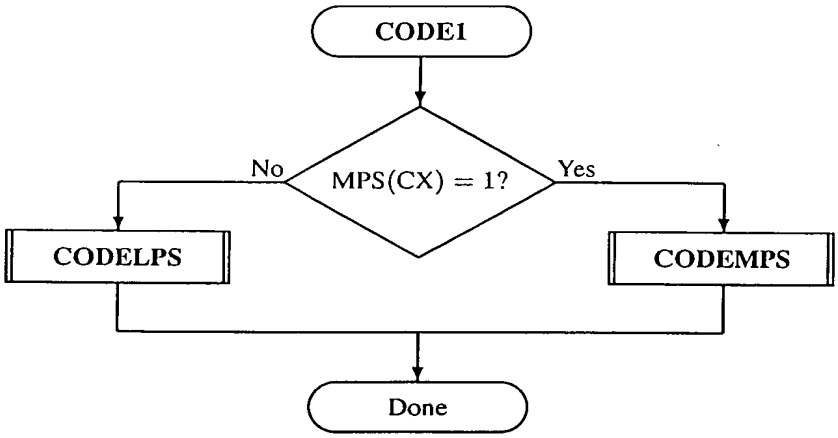

Figure C-4 — CODE1 procedure

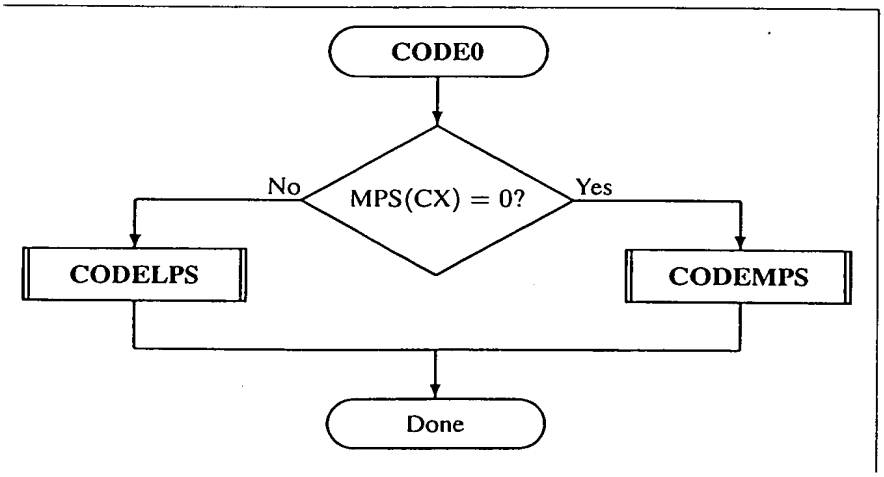

Figure C-5 — CODE0 procedure

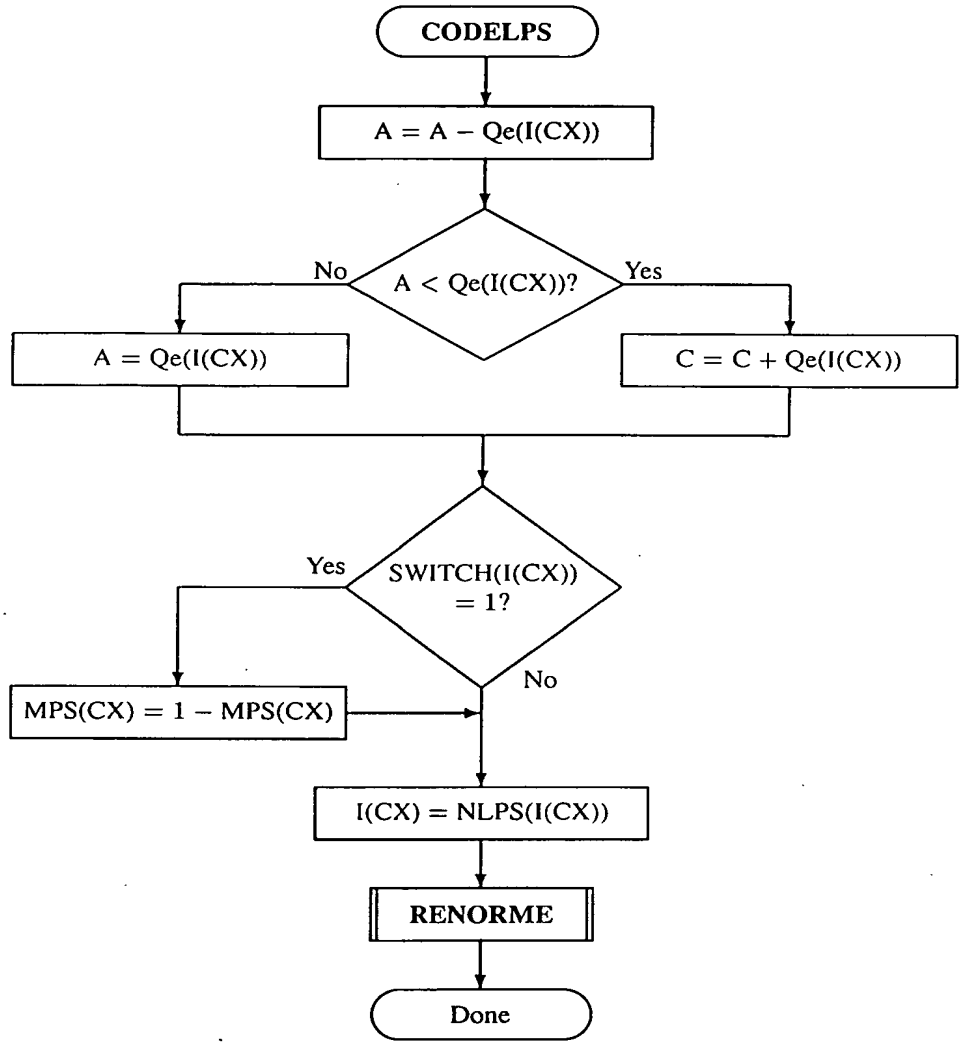

Figure C-6 — CODELPS procedure with conditional MPS/LPS exchange

Table C-2 — Qe values and probability estimation

Index	Qe_Value			NMPS	NLPS	SWITCH
	(hexadecimal)	(binary)	(decimal)			
0	0x5601	0101 0110 0000 0001	0,503 937	1	1	1
1	0x3401	0011 0100 0000 0001	0,304 715	2	6	0
2	0x1801	0001 1000 0000 0001	0,140 650	3	9	0
3	0x0AC1	0000 1010 1100 0001	0,063 012	4	12	0
4	0x0521	0000 0101 0010 0001	0,030 053	5	29	0

Table C-2 — Qe values and probability estimation

Index	Qe_Value			NMPS	NLPS	SWITCH
	(hexadecimal)	(binary)	(decimal)			
5	0x0221	0000 0010 0010 0001	0,012 474	38	33	0
6	0x5601	0101 0110 0000 0001	0,503 937	7	6	1
7	0x5401	0101 0100 0000 0001	0,492 218	8	14	0
8	0x4801	0100 1000 0000 0001	0,421 904	9	14	0
9	0x3801	0011 1000 0000 0001	0,328 153	10	14	0
10	0x3001	0011 0000 0000 0001	0,281 277	11	17	0
11	0x2401	0010 0100 0000 0001	0,210 964	12	18	0
12	0x1C01	0001 1100 0000 0001	0,164 088	13	20	0
13	0x1601	0001 0110 0000 0001	0,128 931	29	21	0
14	0x5601	0101 0110 0000 0001	0,503 937	15	14	1
15	0x5401	0101 0100 0000 0001	0,492 218	16	14	0
16	0x5101	0101 0001 0000 0001	0,474 640	17	15	0
17	0x4801	0100 1000 0000 0001	0,421 904	18	16	0
18	0x3801	0011 1000 0000 0001	0,328 153	19	17	0
19	0x3401	0011 0100 0000 0001	0,304 715	20	18	0
20	0x3001	0011 0000 0000 0001	0,281 277	21	19	0
21	0x2801	0010 1000 0000 0001	0,234 401	22	19	0
22	0x2401	0010 0100 0000 0001	0,210 964	23	20	0
23	0x2201	0010 0010 0000 0001	0,199 245	24	21	0
24	0x1C01	0001 1100 0000 0001	0,164 088	25	22	0
25	0x1801	0001 1000 0000 0001	0,140 650	26	23	0
26	0x1601	0001 0110 0000 0001	0,128 931	27	24	0
27	0x1401	0001 0100 0000 0001	0,117 212	28	25	0
28	0x1201	0001 0010 0000 0001	0,105 493	29	26	0
29	0x1101	0001 0001 0000 0001	0,099 634	30	27	0

Table C-2 — Qe values and probability estimation

Index	Qe_Value			NMPS	NLPS	SWITCH
	(hexadecimal)	(binary)	(decimal)			
30	0x0AC1	0000 1010 1100 0001	0,063 012	31	28	0
31	0x09C1	0000 1001 1100 0001	0,057 153	32	29	0
32	0x08A1	0000 1000 1010 0001	0,050 561	33	30	0
33	0x0521	0000 0101 0010 0001	0,030 053	34	31	0
34	0x0441	0000 0100 0100 0001	0,024 926	35	32	0
35	0x02A1	0000 0010 1010 0001	0,015 404	36	33	0
36	0x0221	0000 0010 0010 0001	0,012 474	37	34	0
37	0x0141	0000 0001 0100 0001	0,007 347	38	35	0
38	0x0111	0000 0001 0001 0001	0,006 249	39	36	0
39	0x0085	0000 0000 1000 0101	0,003 044	40	37	0
40	0x0049	0000 0000 0100 1001	0,001 671	41	38	0
41	0x0025	0000 0000 0010 0101	0,000 847	42	39	0
42	0x0015	0000 0000 0001 0101	0,000 481	43	40	0
43	0x0009	0000 0000 0000 1001	0,000 206	44	41	0
44	0x0005	0000 0000 0000 0101	0,000 114	45	42	0
45	0x0001	0000 0000 0000 0001	0,000 023	45	43	0
46	0x5601	0101 0110 0000 0001	0,503 937	46	46	0

The CODEMPS (Figure C-7) procedure usually reduces the size of the interval to the MPS sub-interval and adjusts the code register so that it points to the base of the MPS sub-interval. However, if the interval sizes are inverted, the LPS sub-interval is coded instead. The size inversion cannot occur unless a renormalization (RENORME) is required after the coding of the symbol. The probability estimate update changes the index I(CX) according to the next MPS index (NMPS) column in Table C-2.

C.2.5 Probability Estimation

Table C-2 shows the Qe value associated with each Qe index. The Qe values are expressed as hexadecimal integers, as binary integers, and as decimal fractions. To convert the 15 bit integer representation of Qe to the decimal probability, the Qe values are divided by (4/3) * (0x8000).

The estimator can be defined as a finite-state machine – a table of Qe indexes and associated next states for each type of renormalization (i.e., new table positions) – as shown in Table C-2. The change in state occurs only when the arithmetic coder interval register is renormalized. This is always done after coding the LPS, and whenever the interval register is less than 0x8000 (0,75 in decimal notation) after coding the MPS.

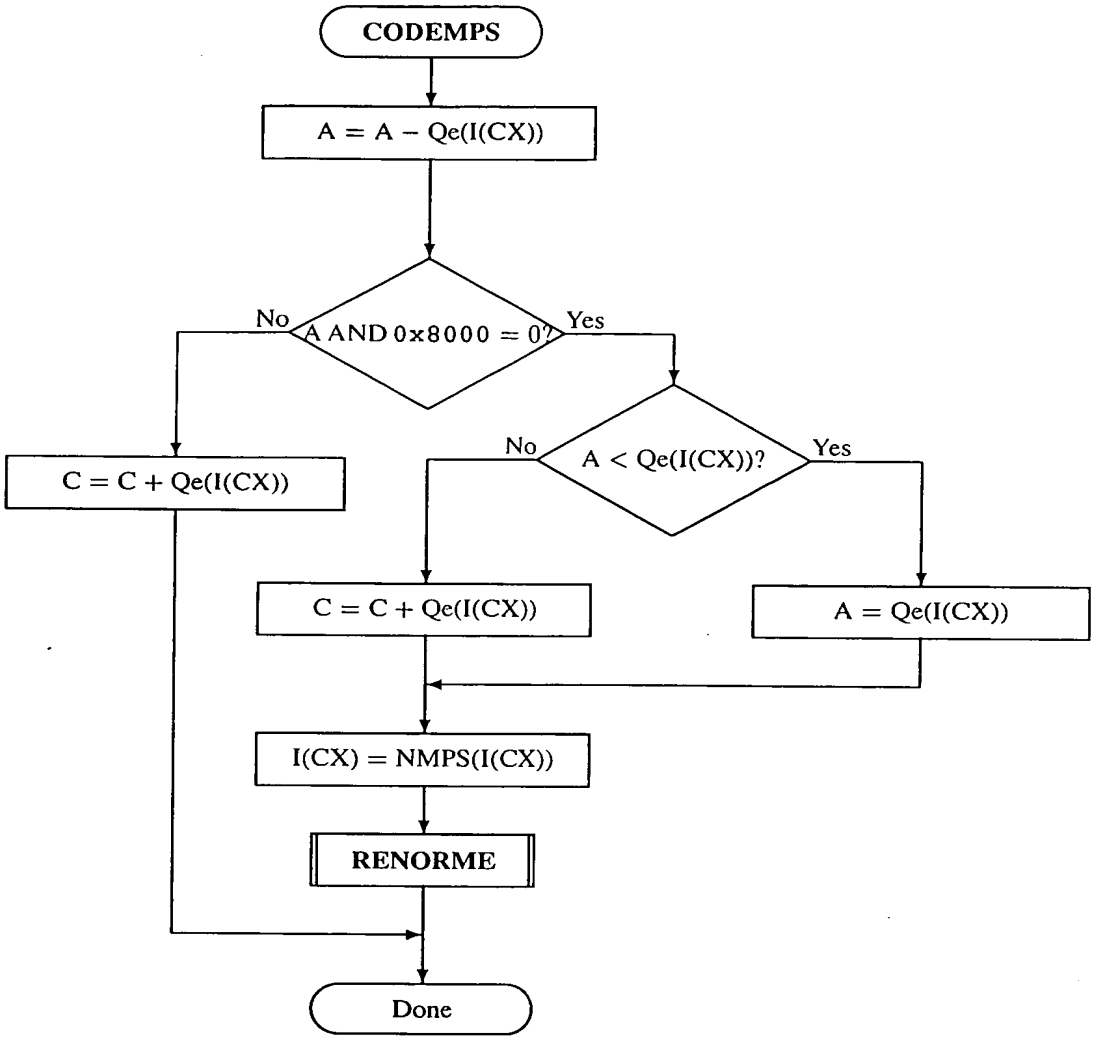

Figure C-7 — CODEMPS procedure with conditional MPS/LPS exchange

After an LPS renormalization, NLPS gives the new index for the LPS probability estimate. If the switch is 1, the MPS symbol sense is reversed.

The index to the current estimate is part of the information stored for context CX. This index is used as the index to the table of values in NMPS, which gives the next index for an MPS renormalization. This index is saved in the context storage at CX. MPS(CX) does not change.

The procedure for estimating the probability on the LPS renormalization path is similar to that of an MPS renormalization, except that when SWITCH(I(CX)) is 1, the sense of MPS(CX) is inverted.

The final index state 46 can be used to establish a fixed 0,5 probability estimate.

C.2.6 Renormalization in the encoder (RENORME) (informative)

Renormalization is very similar in both encoder and decoder, except that in the encoder it generates compressed bits and in the decoder it consumes compressed bits.

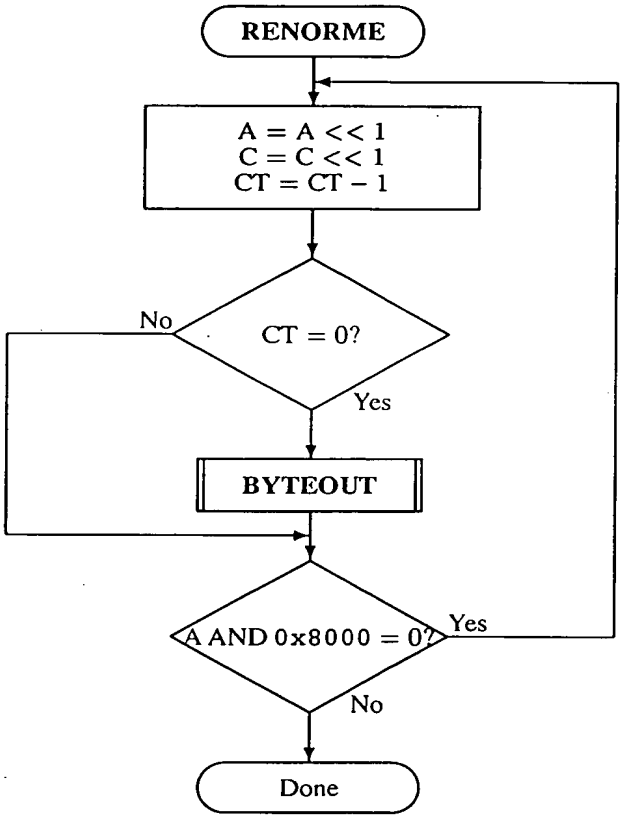

Figure C-8 — Encoder renormalisation procedure

The RENORME procedure for the encoder renormalization is illustrated in Figure C-8. Both the interval register A and the code register C are shifted, one bit at a time. The number of shifts is counted in the counter CT, and when CT is counted down to zero, a byte of compressed image data is removed from C by the procedure BYTEOUT. Renormalization continues until A is no longer less than 0x8000.

C.2.7 Compressed image data output (BYTEOUT) (informative)

The BYTEOUT routine called from RENORME is illustrated in Figure C-9. This routine contains the bit-stuffing procedures which are needed to limit carry propagation into the completed bytes of compressed image data. The conventions used make it impossible for a carry to propagate through more than the byte most recently written to the compressed image data buffer.

The procedure in the block in the lower right section does bit stuffing after a 0xFF byte; the similar procedure on the left is for the case where bit stuffing is not needed.

B is the byte pointed to by the compressed image data buffer pointer BP. If B is not a 0xFF byte, the carry bit is checked. If the carry bit is set, it is added to B and B is again checked to see if a bit needs to be stuffed in the next byte. After the need for bit stuffing has been determined, the appropriate path is chosen, BP is incremented and the new value of B is removed from the code register "b" bits.

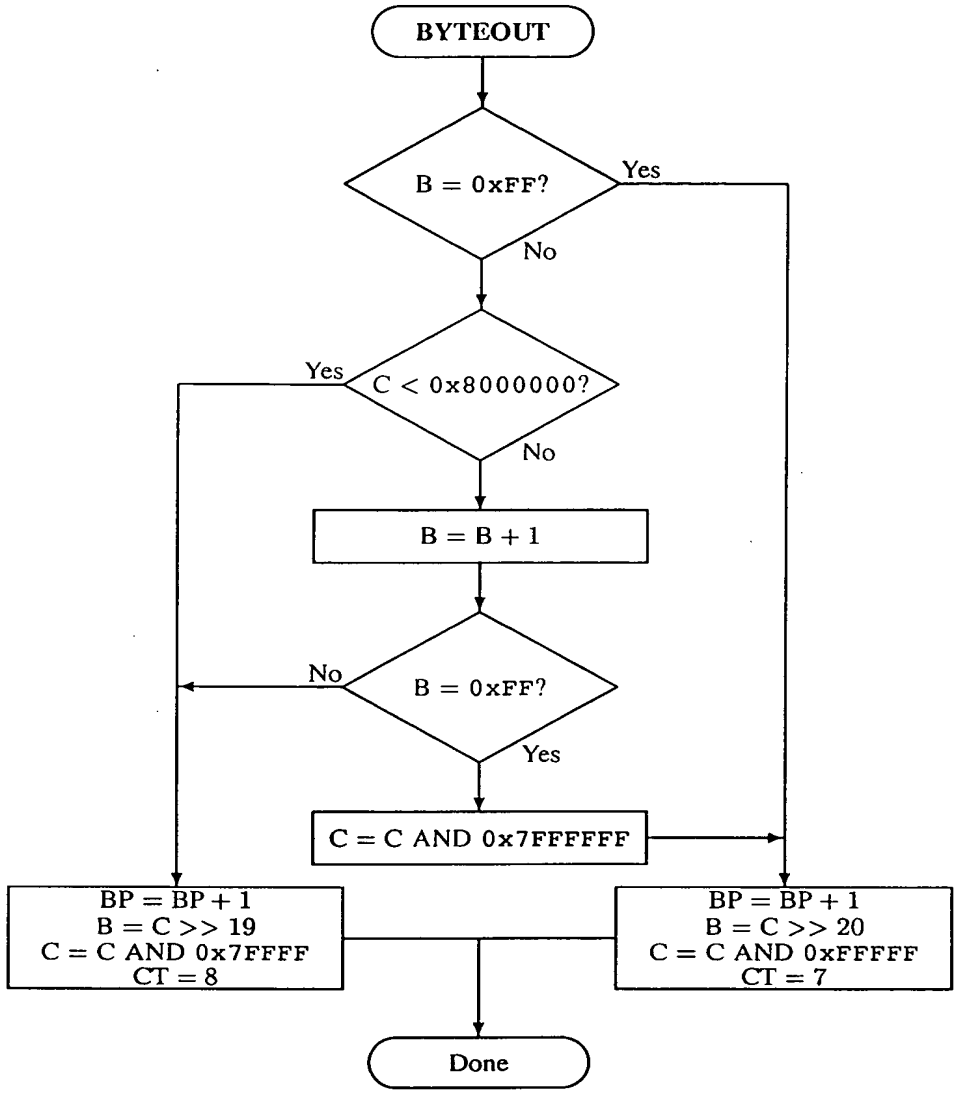

Figure C-9 — BYTEOUT procedure for encoder

C.2.8 Initialisation of the encoder (INITENC) (informative)

The INITENC procedure is used to start the arithmetic coder. After MPS and I are initialized, the basic steps are shown in Figure C-10.

The interval register and code register are set to their initial values, and the bit counter is set. Setting CT = 12 reflects the fact that there are three spacer bits in the register which need to be filled before the field from which the bytes are removed is reached. BP always points to the byte preceding the position BPST where the first byte is placed. Therefore, if the preceding byte is a 0xFF byte, a spurious bit stuff will occur, but can be compensated for by increasing CT. The initial settings for MPS and I are shown in Table D-7.

C.2.9 Termination of coding (FLUSH) (informative)

The FLUSH procedure shown in Figure C-11 is used to terminate the encoding operations and generate the required terminating marker. The procedure guarantees that the 0xFF prefix to the marker code overlaps the final bits of the

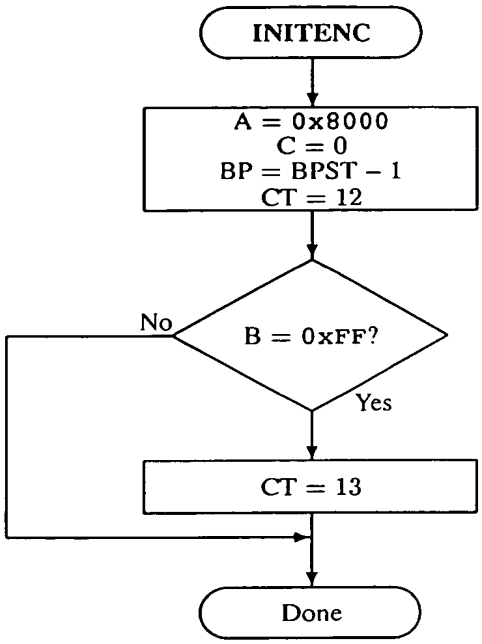

Figure C-10 — Initialisation of the encoder

compressed image data. This guarantees that any marker code at the end of the compressed image data will be recognized and interpreted before decoding is complete.

The first part of the FLUSH procedure sets as many bits in the C-register to 1 as possible as shown in Figure C-12. The exclusive upper bound for the C-register is the sum of the C-register and the interval register. The low order 16 bits of C are forced to 1, and the result is compared to the upper bound. If C is too big, the leading 1-bit is removed, reducing C to a value which is within the interval.

The byte in the C-register is then completed by shifting C, and two bytes are then removed. If the byte in buffer, B, is an 0xFF then it is discarded. Otherwise, buffer B is output to the bit stream.

NOTE — This is the only normative option for termination in ITU-T Rec.T.88 | ISO/IEC 14492. However, further reduction of the bit stream is allowed in this Recommendation | International Standard provided correct decoding is assured (see Table D.4.2).

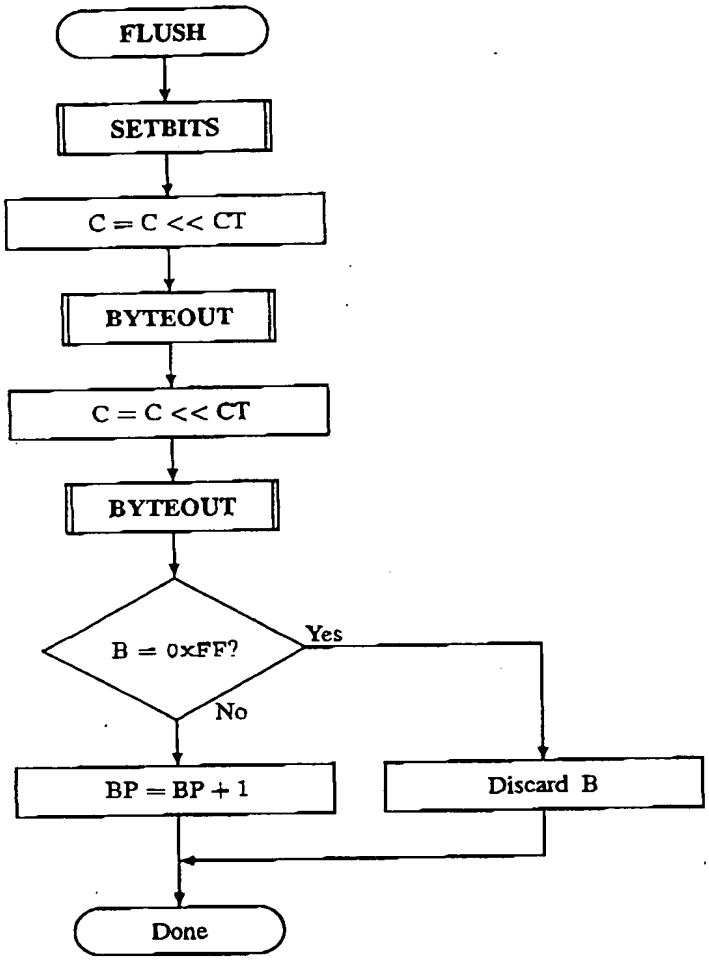

Figure C-11 — FLUSH procedure

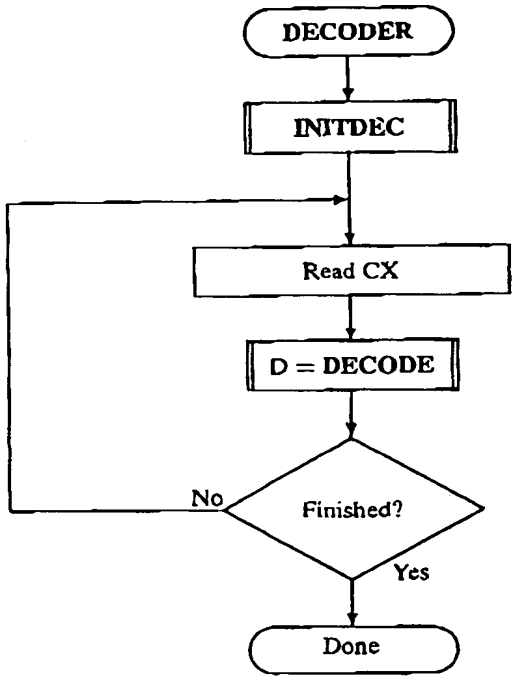

Figure C-14 — Decoder for the MQ-coder

C.3.1 Decoder code register conventions

The flow charts given in this Annex assume the register structures for the decoder shown in Table C-3.

Table C-3 — Decoder register structures

	MSB	LSB
Chigh register	xxxx xxxx	xxxx xxxx
Clow register	bbbb bbbb	0000 0000
A-register	aaaa aaaa	aaaa aaaa

Chigh and Clow can be thought of as one 32 bit C-register in that renormalization of C shifts a bit of new data from the MSB of Clow to the LSB of Chigh. However, the decoding comparisons use Chigh alone. New data is inserted into the "b" bits of Clow one byte at a time.

The detailed description of the handling of data with stuff-bits will be given later in this Annex.

Note that the comparisons shown in the various procedures in this section assume precisions greater than 16 bits. Logical comparisons can be used with 16 bit precision.

C.3.2 Decoding a decision (DECODE)

The decoder decodes one binary decision at a time. After decoding the decision, the decoder subtracts any amount from the compressed image data that the encoder added. The amount left in the compressed image data is the offset from the

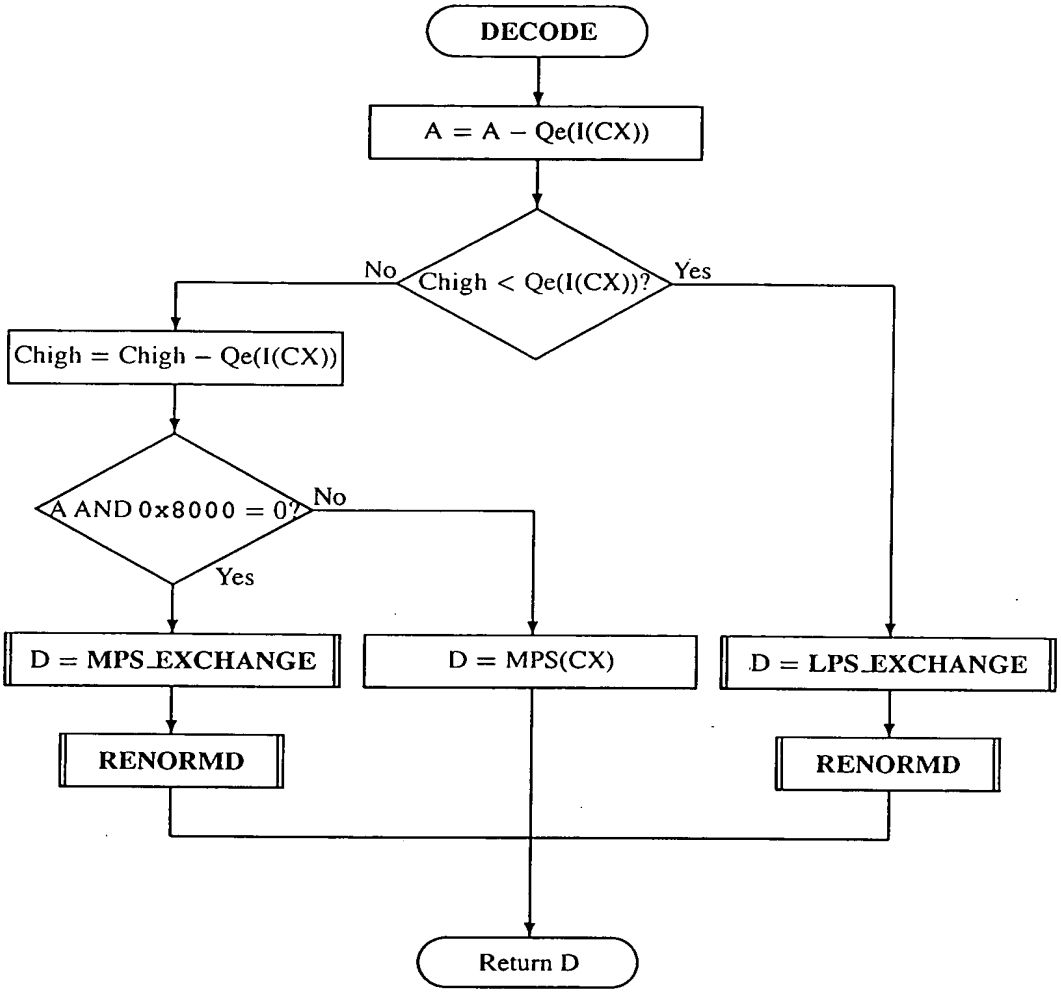

Figure C-15 — Decoding an MPS or an LPS

base of the current interval to the sub-interval allocated to all binary decisions not yet decoded. In the first test in the DECODE procedure illustrated in Figure C-15 the Chigh register is compared to the size of the LPS sub-interval. Unless a conditional exchange is needed, this test determines whether a MPS or LPS is decoded. If Chigh is logically greater than or equal to the LPS probability estimate Qe for the current index I stored at CX, then Chigh is decremented by that amount. If A is not less than 0x8000, then the MPS sense stored at CX is used to set the decoded decision D.

When a renormalization is needed, the MPS/LPS conditional exchange may have occurred. For the MPS path the conditional exchange procedure is shown in Figure C-16. As long as the MPS sub-interval size A calculated as the first step in Figure C-16 is not logically less than the LPS probability estimate Qe(I(CX)), an MPS did occur and the decision can be set from MPS(CX). Then the index I(CX) is updated from the next MPS index (NMPS) column in Table C-2. If, however, the LPS sub-interval is larger, the conditional exchange occurred and an LPS occurred. D is set by inverting MPS(CX). The probability update switches the MPS sense if the SWITCH column has a "1" and updates the index I(CX) from the next LPS index (NLPS) column in Table C-2. The probability estimation in the decoder needs to be identical to the probability estimation in the encoder.

For the LPS path of the decoder the conditional exchange procedure is given the LPS_EXCHANGE procedure shown in Figure C-17. The same logical comparison between the MPS sub-interval A and the LPS sub-interval Qe(I(CX)) determines if a conditional exchange occurred. On both paths the new sub-interval A is set to Qe(I(CX)). On the left path

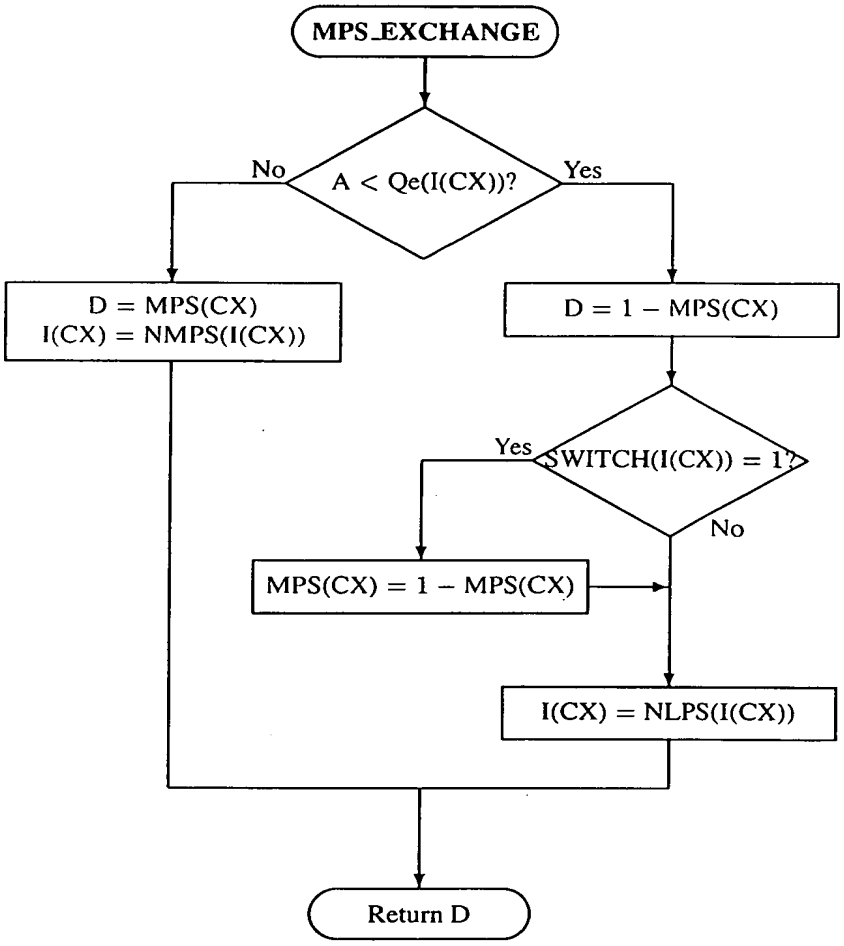

Figure C-16 — Decoder MPS path conditional exchange procedure

the conditional exchange occurred so the decision and update are for the MPS case. On the right path, the LPS decision and update are followed.

C.3.3 Renormalization in the decoder (RENORMD)

The RENORMD procedure for the decoder renormalization is illustrated in Figure C-18. A counter keeps track of the number of compressed bits in the Clow section of the C-register. When CT is zero, a new byte is inserted into Clow in the BYTEIN procedure. The C-register in this procedure is the concatenation of the Chigh and Clow registers.

Both the interval register A and the code register C are shifted, one bit at a time, until A is no longer less than 0x8000.

C.3.4 Compressed image data input (BYTEIN)

The BYTEIN procedure called from RENORMD is illustrated in Figure C-19. This procedure reads in one byte of data, compensating for any stuff bits following the 0xFF byte in the process. It also detects the marker codes which must occur at the end of a coding pass. The C-register in this procedure is the concatenation of the Chigh and Clow registers.

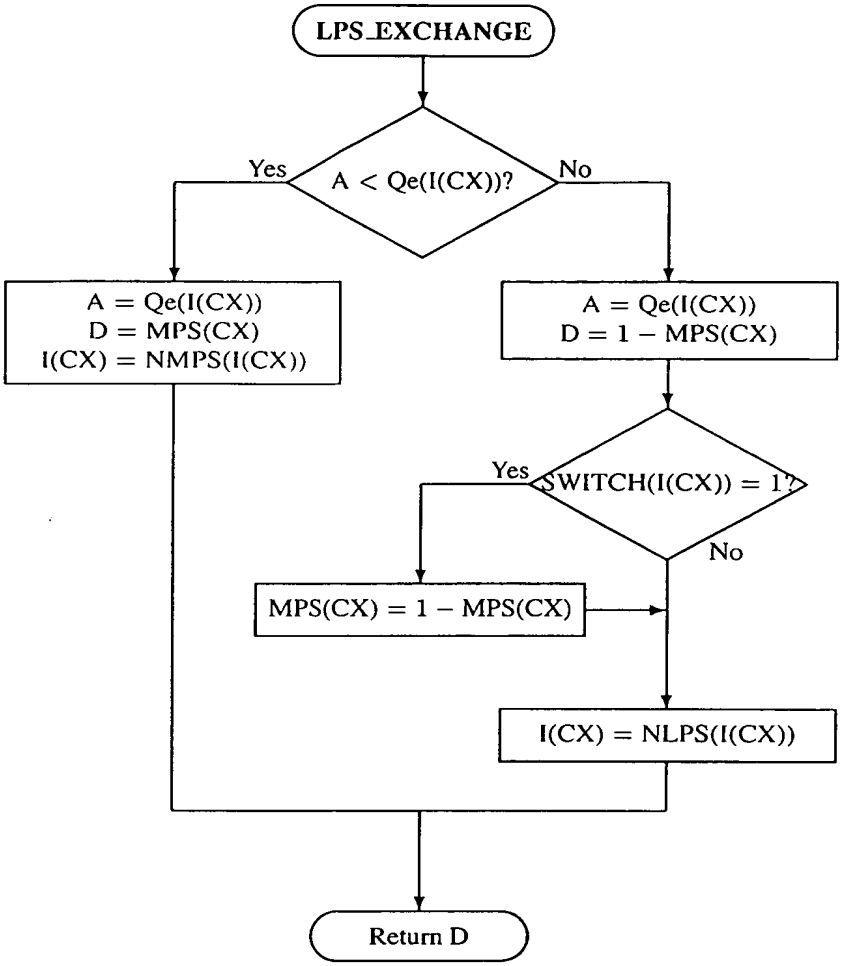

Figure C-17 — Decoder LPS path conditional exchange procedure

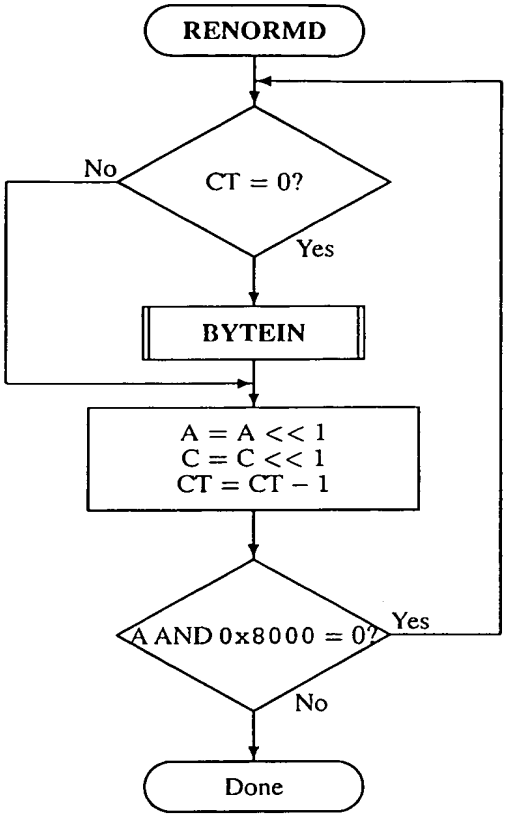

Figure C-18 — Decoder renormalisation procedure

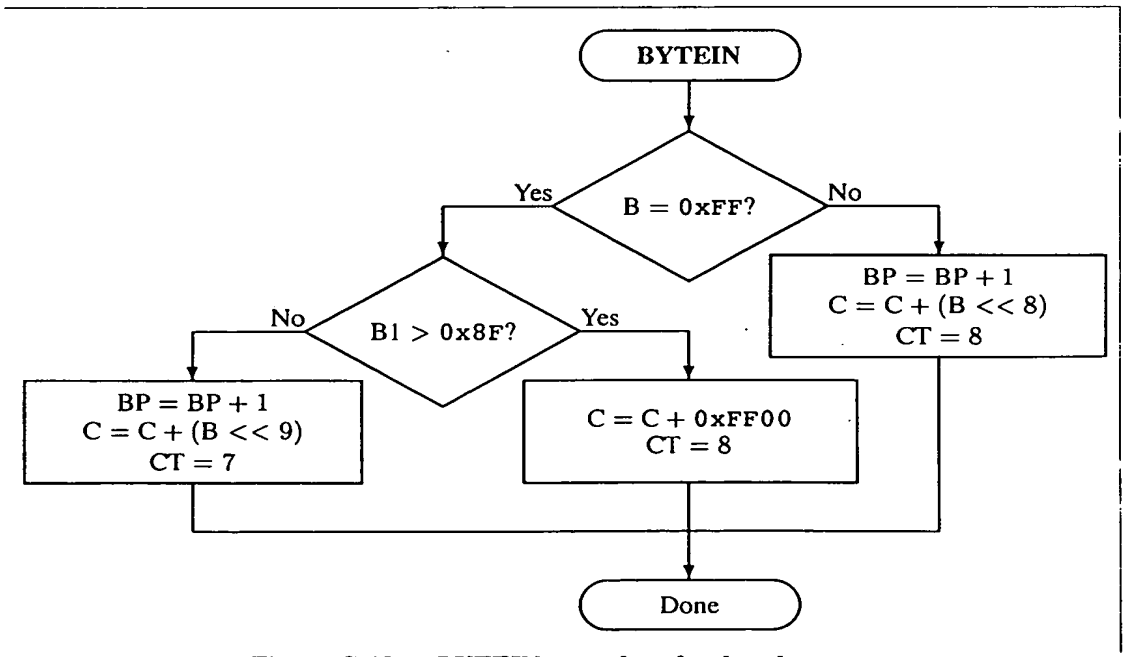

Figure C-19 — BYTEIN procedure for decoder

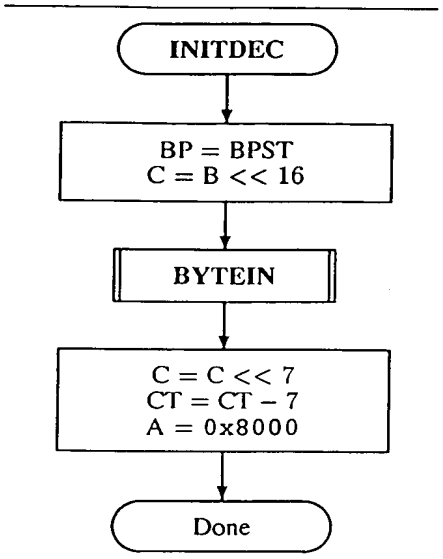

Figure C-20 — Initialisation of the decoder

B is the byte pointed to by the compressed image data buffer pointer BP. If B is not a 0xFF byte, BP is incremented and the new value of B is inserted into the high order 8 bits of Clow.

If B is a 0xFF byte, then B1 (the byte pointed to by BP+1) is tested. If B1 exceeds 0x8F, then B1 must be one of the marker codes. The marker code is interpreted as required, and the buffer pointer remains pointed to the 0xFF prefix of the marker code which terminates the arithmetically compressed image data. 1-bits are then fed to the decoder until the decoding is complete. This is shown by adding 0xFF00 to the C-register and setting the bit counter CT to 8.

If B1 is not a marker code, then BP is incremented to point to the next byte which contains a stuffed bit. The B is added to the C-register with an alignment such that the stuff bit (which contains any carry) is added to the low order bit of Chigh.

C.3.5 Initialisation of the decoder (INITDEC)

The INITDEC procedure is used to start the arithmetic decoder. After MPS and I are initialized, the basic steps are shown in Figure C-20.

BP, the pointer to the compressed image data, is initialized to BPST (pointing to the first compressed byte). The first byte of the compressed image data is shifted into the low order byte of Chigh, and a new byte is then read in. The C-register is then shifted by 7 bits and CT is decremented by 7, bringing the C-register into alignment with the starting value of A. The interval register A is set to match the starting value in the encoder. The initial settings for MPS and I are shown in Table D-7.

C.3.6 Resetting arithmetic coding statistics

At certain points during the decoding some or all of the arithmetic coding statistics are reset. This process involves returning I(CX) and MPS(CX) to their initial values as defined in Table D-7 for some or all values of CX.

C.3.7 Saving arithmetic coding statistics

In some cases, the decoder needs to save or restore some values of I(CX) and MPS(CX).

The claims defining the invention are as follows:

1. An apparatus for JPEG 2000 entropy coding, said apparatus including:

a context generator for generating a context for each bit of one or more coefficients in a code block;

an arithmetic coder for entropy coding each bit to be coded from said code block using said context for said bit; and

a FIFO coupled between said context generator and said arithmetic coder for streamlining transfer of data between said context generator and said arithmetic coder, said FIFO adapted to store each bit, said corresponding context and a repeat number of said bit and context pair.

2. The apparatus according to claim 1, wherein said context generator includes means for generating a repeat pattern of two or more bit and context pairs in a single clock cycle.

3. The apparatus according to claim 2, wherein a run length repeat command represents said repeat pattern.

4. The apparatus according to claim 3, wherein said FIFO stores said run length repeat command as said repeat number.

5. The apparatus according to claim 1, wherein said context generator provides context at variable rates.

6. The apparatus according to claim 1, wherein said arithmetic coder includes means for accelerating coding of a codestream using said repeat pattern.

7. The apparatus according to claim 6, wherein said arithmetic coder further includes means for calculating a repeat count r for two or more bits dependent upon an interval A and a current estimate of LPS probability Qe(I(CX)), where I(CX) is an index stored for a context CX.

8. The apparatus according to claim 7, wherein said arithmetic coder further includes:

means for entropy encoding said two or more bits in a Run Length context using one of said repeat count r and said repeat number dependent upon whether said repeat count r is greater than said repeat number.

9. The apparatus according to claim 1, wherein said FIFO effects a speedup in processing of a first cleanup pass involving run length coding.

10. A method of JPEG 2000 entropy coding, said method including the steps of:

generating a context for each bit of one or more coefficients in a code block;

arithmetic coding each bit to be coded from said code block using said context for said bit; and

buffering using a FIFO to streamline transfer of data between said context generating step and said arithmetic coding step, said FIFO adapted to store each bit, said corresponding context and a repeat number of said bit and context pair.

11. The method according to claim 10, wherein said context generating step includes the step of generating a repeat pattern of two or more bit and context pairs in a single clock cycle.

12. The method according to claim 11, wherein a run length repeat command represents said repeat pattern.

13. The method according to claim 12, wherein said FIFO stores said run length repeat command as said repeat number.

14. The method according to claim 10, wherein said context generating step provides context at variable rates.

15. The method according to claim 10, wherein said arithmetic coding step includes accelerating coding of a codestream using said repeat pattern.

16. The method according to claim 15, wherein said arithmetic coding step further includes the step of calculating a repeat count r for two or more bits dependent upon an interval A and a current estimate of LPS probability Qe(I(CX)), where I(CX) is an index stored for a context CX.

17. The method according to claim 16, wherein said arithmetic coding step further includes the step of entropy encoding said two or more bits in a Run Length context using one of said repeat count r and said repeat number dependent upon whether said repeat count r is greater than said repeat number.

18. The method according to claim 10, wherein said FIFO effects a speedup in processing of a first cleanup pass involving run length coding.

19. A computer program product having a computer readable medium having a computer program recorded therein for JPEG 2000 entropy coding, said computer program product including:

computer program code means for generating a context for each bit of one or more coefficients in a code block;

computer program code means for arithmetic coding each bit to be coded from said code block using said context for said bit; and

computer program code means for providing a FIFO between said context generating and said arithmetic coding to streamline transfer of data between said context generator and said arithmetic coder, said FIFO adapted to store each bit, said corresponding context and a repeat number of said bit and context pair.

20. The computer program product according to claim 19, wherein said computer program code means for context generating includes computer program code means for generating a repeat pattern of two or more bit and context pairs in a single clock cycle.

21. The computer program product according to claim 20, wherein a run length repeat command represents said repeat pattern.

22. The computer program product according to claim 21, wherein said FIFO stores said run length repeat command as said repeat number.

23. The computer program product according to claim 19, wherein said computer program code means for context generating provides context at variable rates.

24. The computer program product according to claim 19, wherein said computer program code means for arithmetic coding includes computer program code means for accelerating coding of a codestream using said repeat pattern.

25. The computer program product according to claim 24, wherein said computer program code means for arithmetic coding further includes computer program code means for calculating a repeat count r for two or more bits dependent upon an interval A and a current estimate of LPS probability Qe(I(CX)), where I(CX) is an index stored for a context CX.

26. The computer program product according to claim 25, wherein said computer program code means for arithmetic coding further includes:

computer program code means for entropy encoding said two or more bits in a Run Length context using one of said repeat count r and said repeat number dependent upon whether said repeat count r is greater than said repeat number.

27. An apparatus for JPEG 2000 entropy coding substantially as hereinbefore described with reference to Figs. 1-28 of the accompanying drawings.

28. A method of JPEG 2000 entropy coding substantially as hereinbefore described with reference to Figs. 1-28 of the accompanying drawings.

29. A computer program product having a computer readable medium having a computer program recorded therein for JPEG 2000 entropy coding substantially as hereinbefore described with reference to Figs. 1-28 of the accompanying drawings.

DATED this Sixteenth Day of November, 2000
Canon Kabushiki Kaisha
Patent Attorneys for the Applicant
SPRUSON & FERGUSON

APPARATUS EFFECTING IMPROVED ARITHMETIC CODING IN JPEG 2000 ENTROPY CODER

ABSTRACT

A method, an apparatus (100) and a computer program product for JPEG entropy coding are disclosed. In the method, transform coefficients of a code block (200) in sign-magnitude form are pre-analyzed. Statistical data about the coefficients is stored, preferably with the coefficients. Significance state data (330), coded data (340), magnitude refinement data (350), bit data (310), and sign data (320) for the code block are buffered. More preferably, the buffering is implemented using a FIFO (120, 1130), located between a context generation module (110, 1120) and an arithmetic coder (130, 1150). Based upon the statistical data, at least one command for at least one sequence of bit and context pairs is generated for arithmetic encoding.

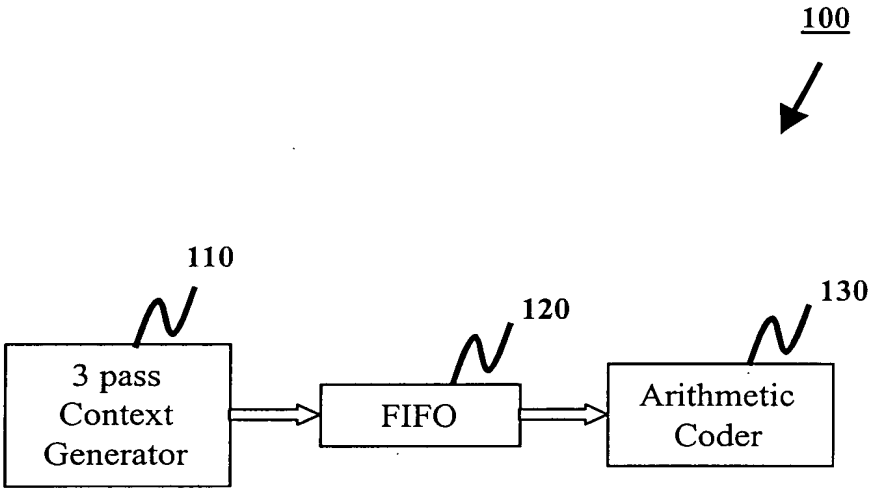
100
110
120
130
3 pass Context Generator
FIFO
Arithmetic Coder

FIG. 1

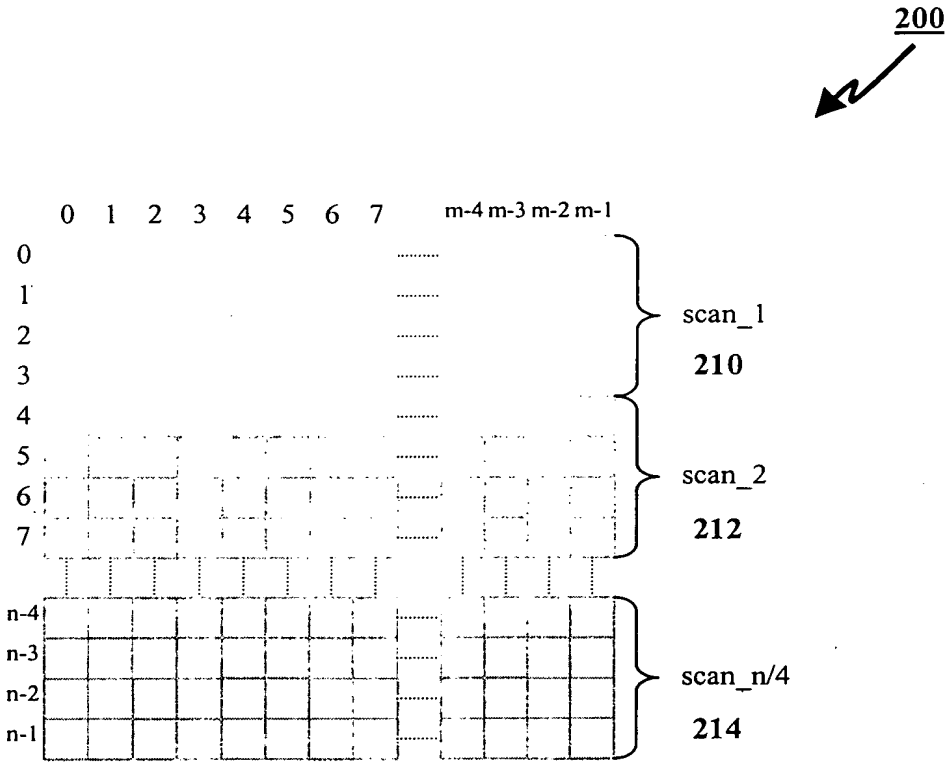
200
0 1 2 3 4 5 6 7
m-4 m-3 m-2 m-1
0
1
2
3
4
5
6
7
n-4
n-3
n-2
n-1
scan_1
210
scan_2
212
scan_n/4
214

FIG. 2

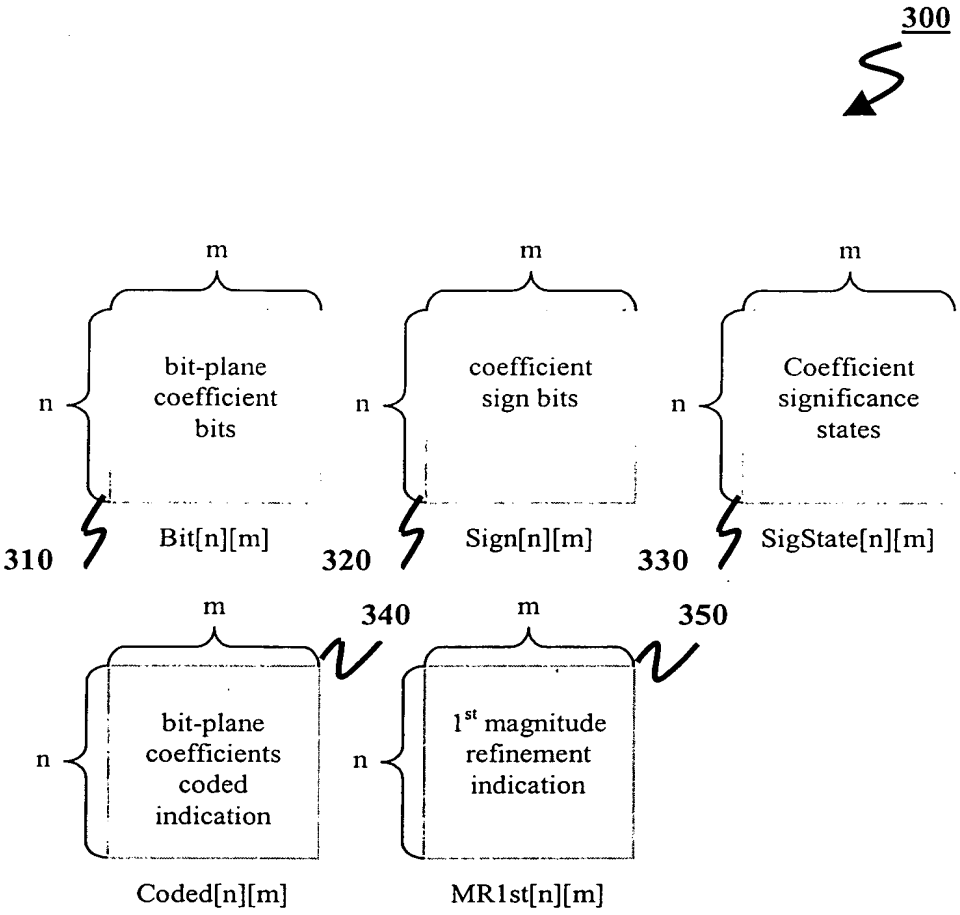
300
m
n
bit-plane coefficient bits
Bit[n][m]
310
m
n
coefficient sign bits
Sign[n][m]
320
m
n
Coefficient significance states
SigState[n][m]
330
m
340
n
bit-plane coefficients coded indication
Coded[n][m]
m
350
n
1st magnitude refinement indication
MR1st[n][m]

FIG. 3

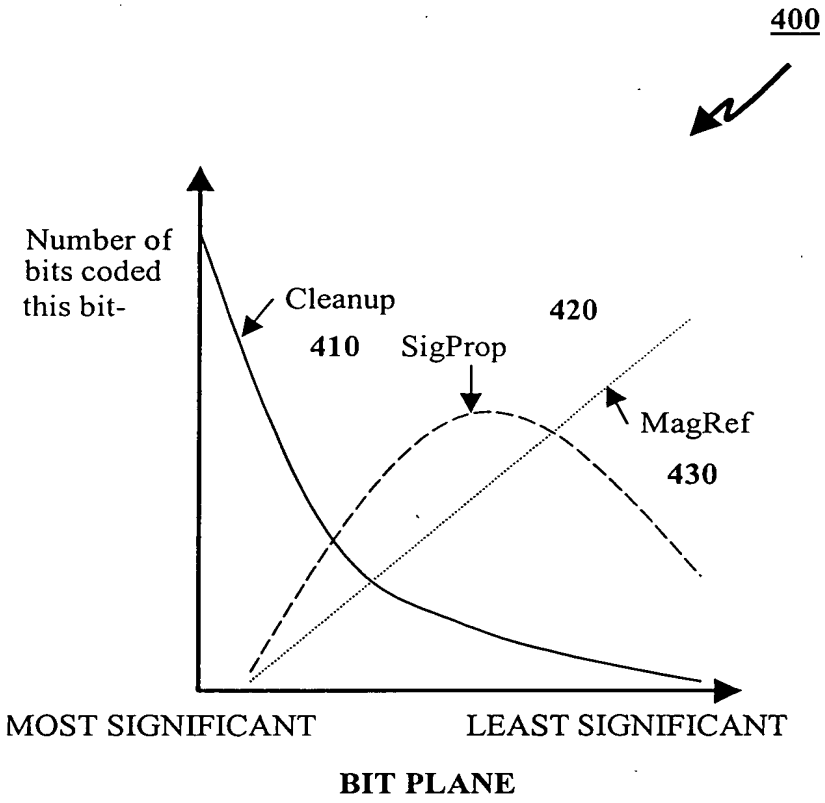
400
Number of bits coded this bit-
Cleanup
410
SigProp
420
MagRef
430
MOST SIGNIFICANT
LEAST SIGNIFICANT
BIT PLANE

FIG. 4

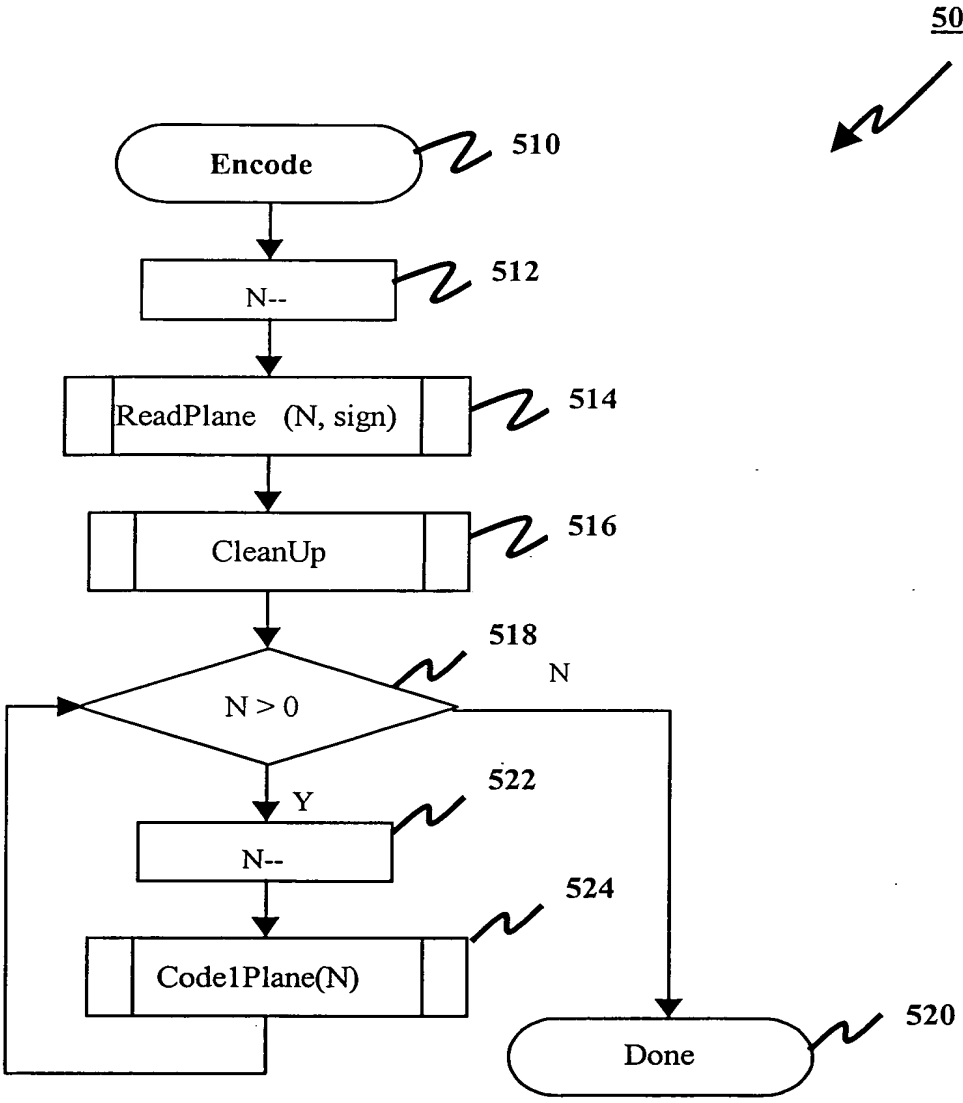
500
Encode
510
N--
512
ReadPlane (N, sign)
514
CleanUp
516
518
N > 0
N
Y
522
N--
524
Code1Plane(N)
Done
520

FIG. 5

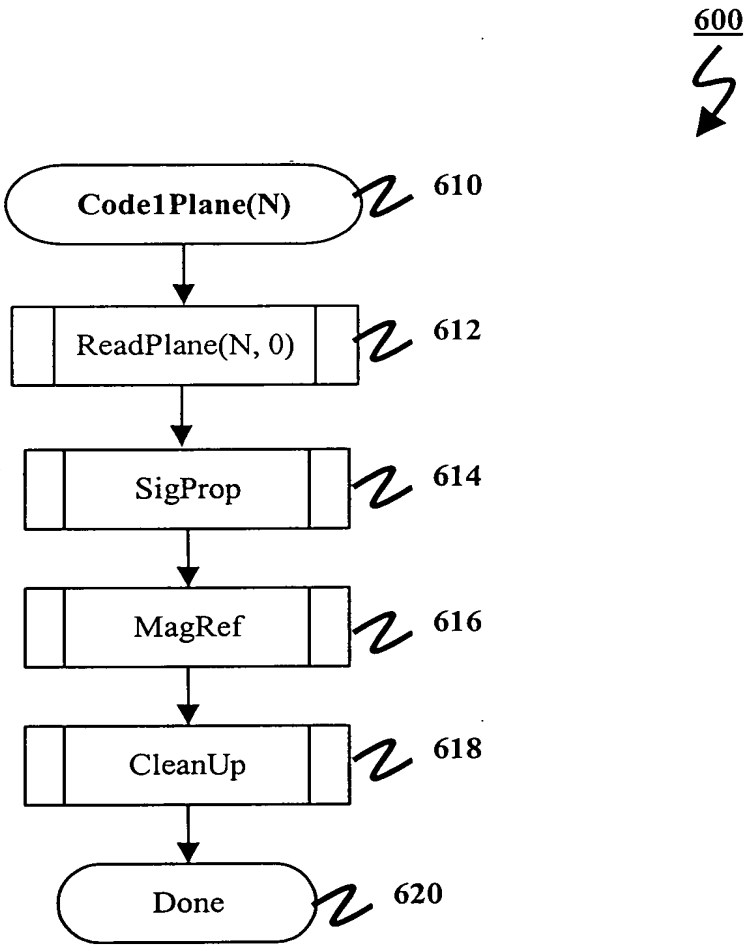
600
Code1Plane(N)
610
ReadPlane(N, 0)
612
SigProp
614
MagRef
616
CleanUp
618
Done
620

FIG. 6

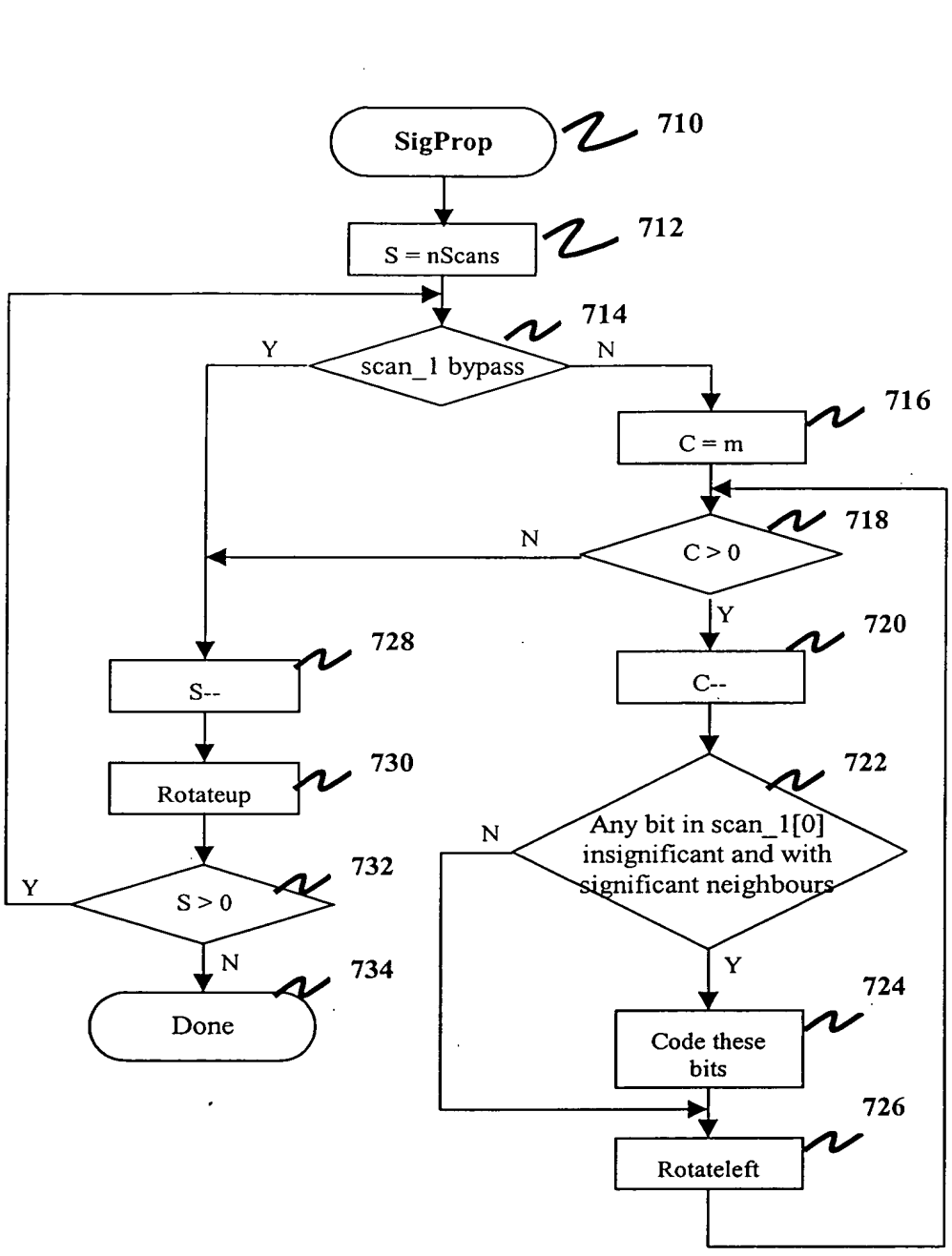
700
SigProp
710
S = nScans
712
714
scan_1 bypass
Y
N
716
C = m
718
C > 0
N
Y
720
C--
722
Any bit in scan_1[0] insignificant and with significant neighbours
N
Y
724
Code these bits
726
Rotateleft
728
S--
730
Rotateup
732
S > 0
Y
N
734
Done

FIG. 7

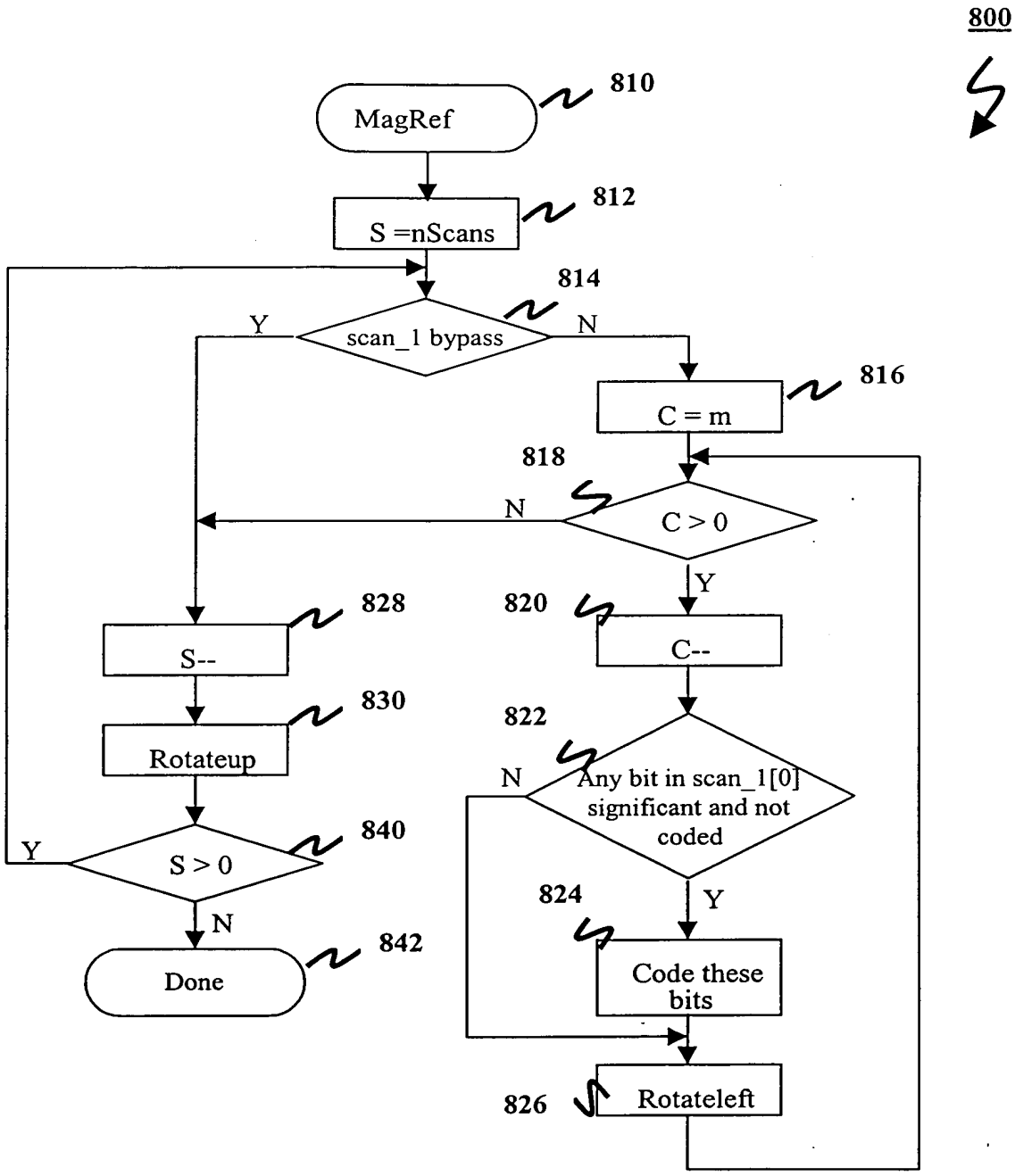
800
MagRef
810
S =nScans
812
scan_1 bypass
814
Y
N
C = m
816
C > 0
818
N
Y
S--
828
C--
820
Rotateup
830
Any bit in scan_1[0] significant and not coded
822
N
S > 0
840
Y
N
Done
842
Code these bits
824
Y
Rotateleft
826

FIG. 8

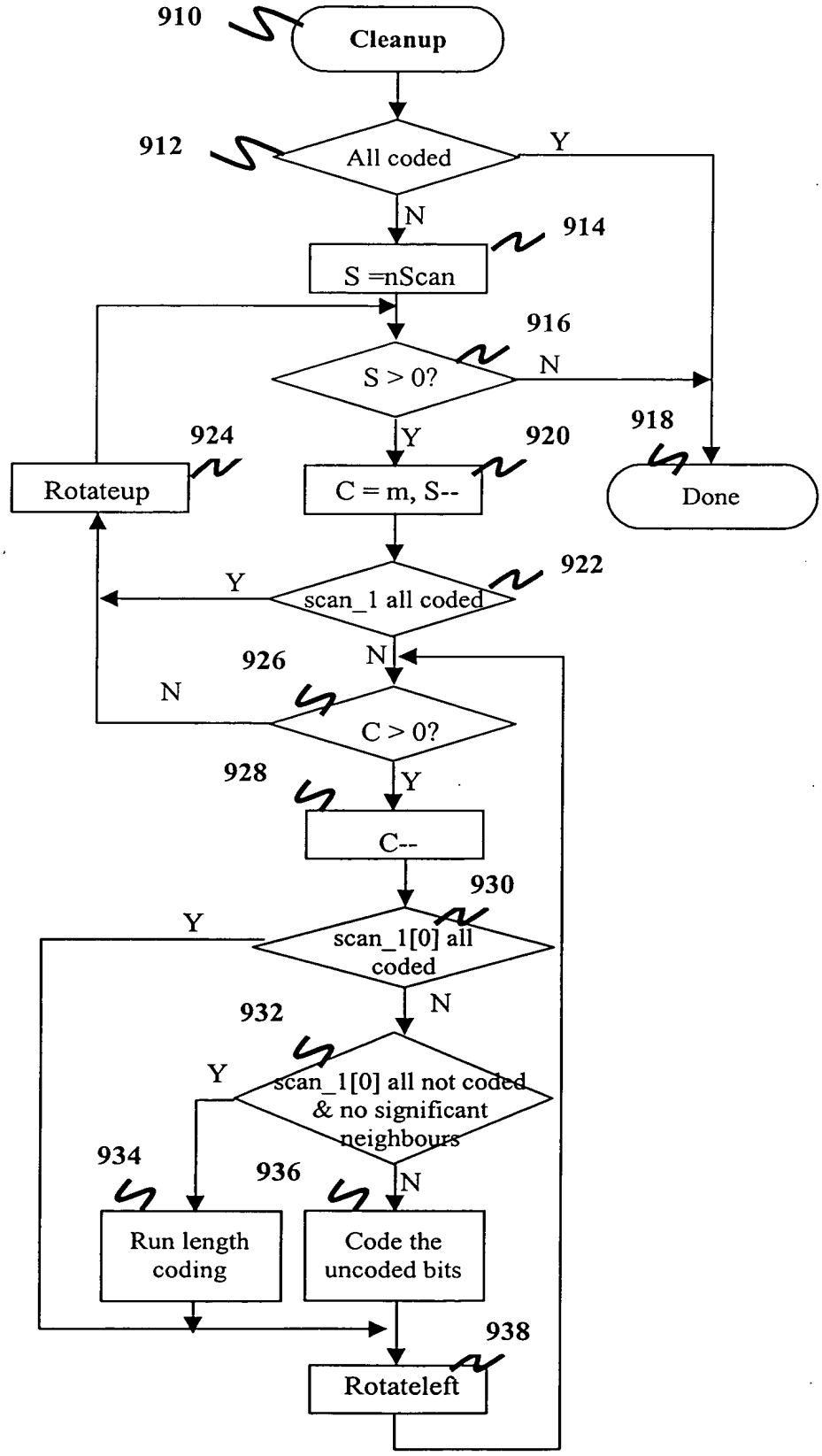
900
910
Cleanup
912
All coded
Y
N
914
S =nScan
916
S > 0?
N
Y
918
Done
920
C = m, S--
924
Rotateup
922
scan_1 all coded
Y
N
926
C > 0?
N
928
Y
C--
930
scan_1[0] all coded
Y
N
932
scan_1[0] all not coded & no significant neighbours
Y
N
934
Run length coding
936
Code the uncoded bits
938
Rotateleft

FIG. 9

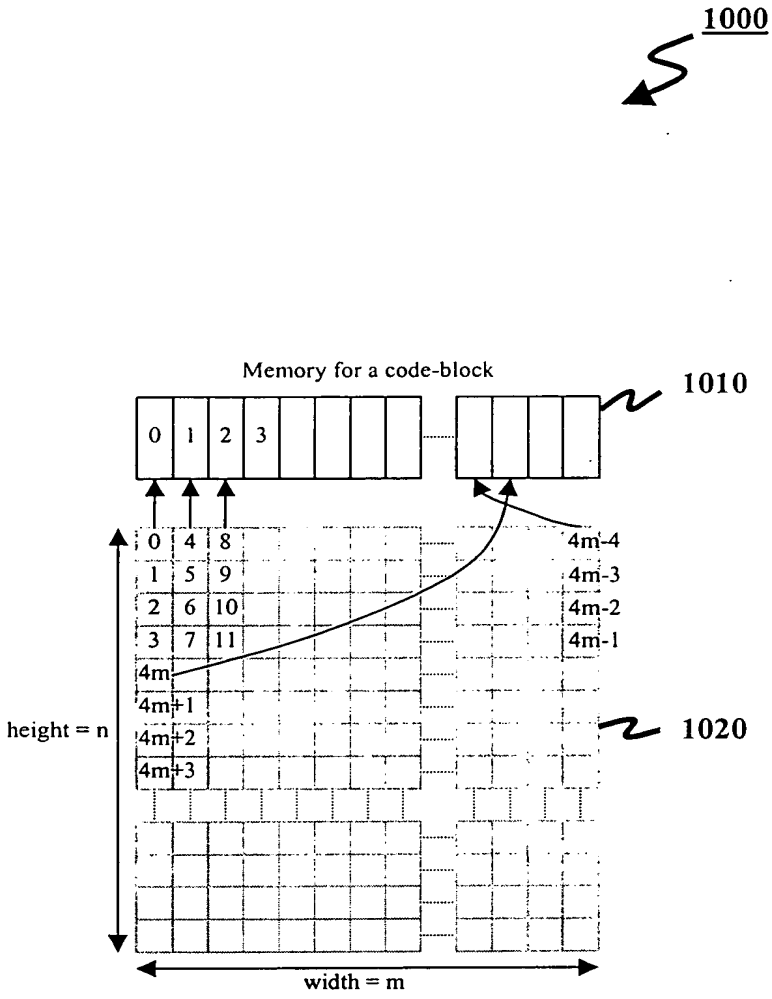
1000
Memory for a code-block
1010
0
1
2
3
0
4
8
1
5
9
2
6
10
3
7
11
4m
4m+1
4m+2
4m+3
4m-4
4m-3
4m-2
4m-1
1020
height = n
width = m

FIG. 10

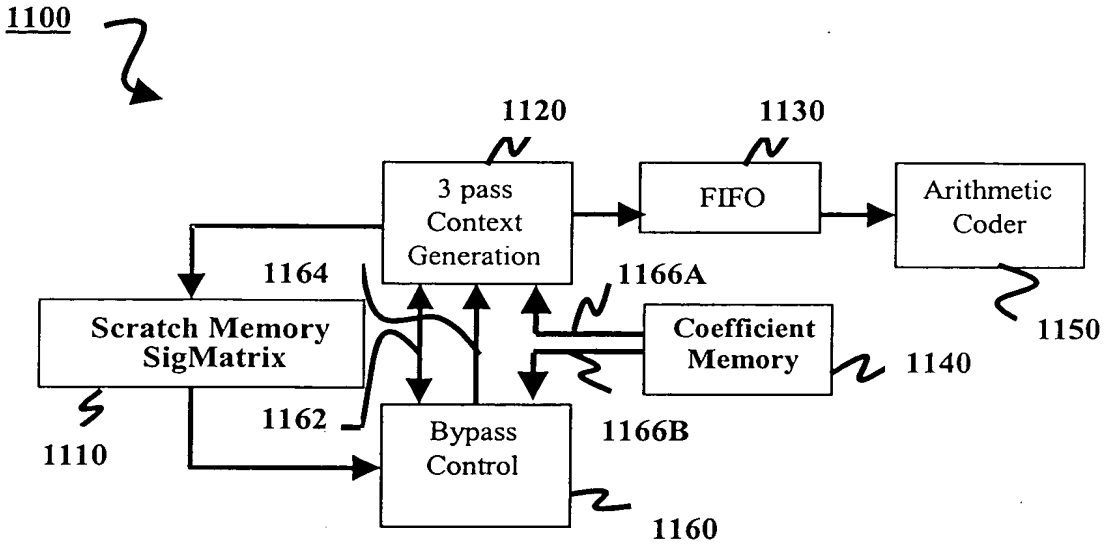
1100
1120
1130
3 pass Context Generation
FIFO
Arithmetic Coder
1164
1166A
Scratch Memory SigMatrix
Coefficient Memory
1150
1140
1110
1162
Bypass Control
1166B
1160

FIG. 11

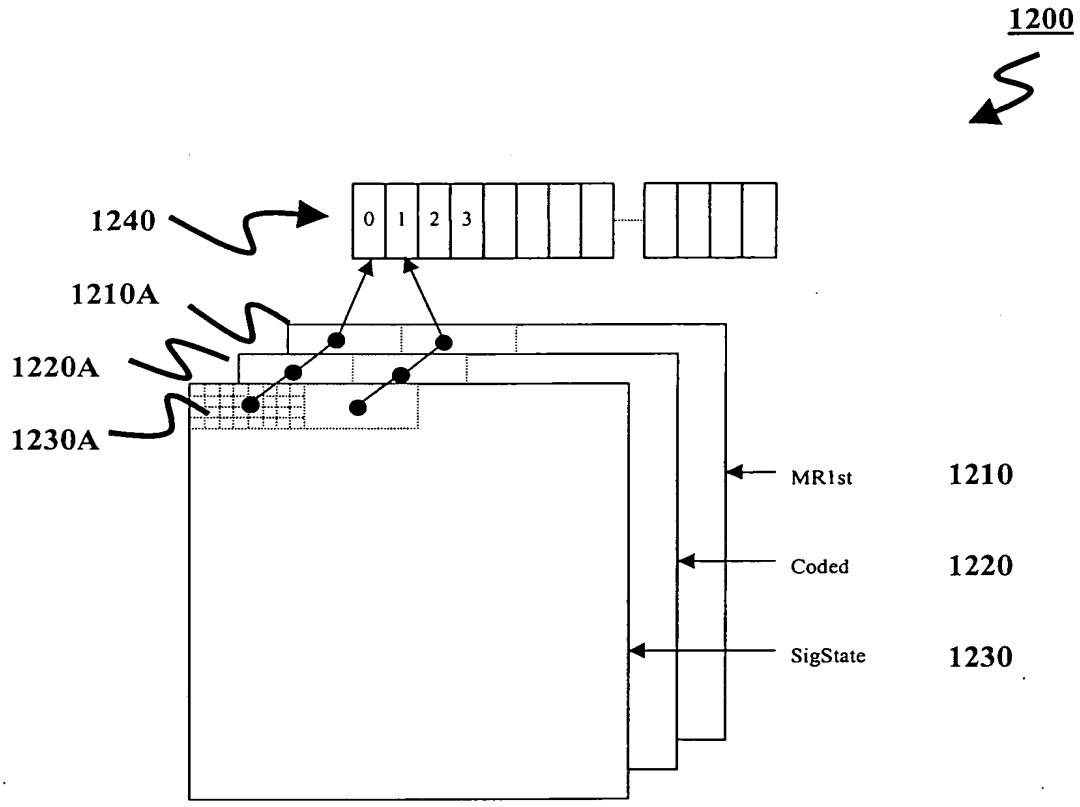
1200
1240
0
1
2
3
1210A
1220A
1230A
MR1st
1210
Coded
1220
SigState
1230

FIG. 12

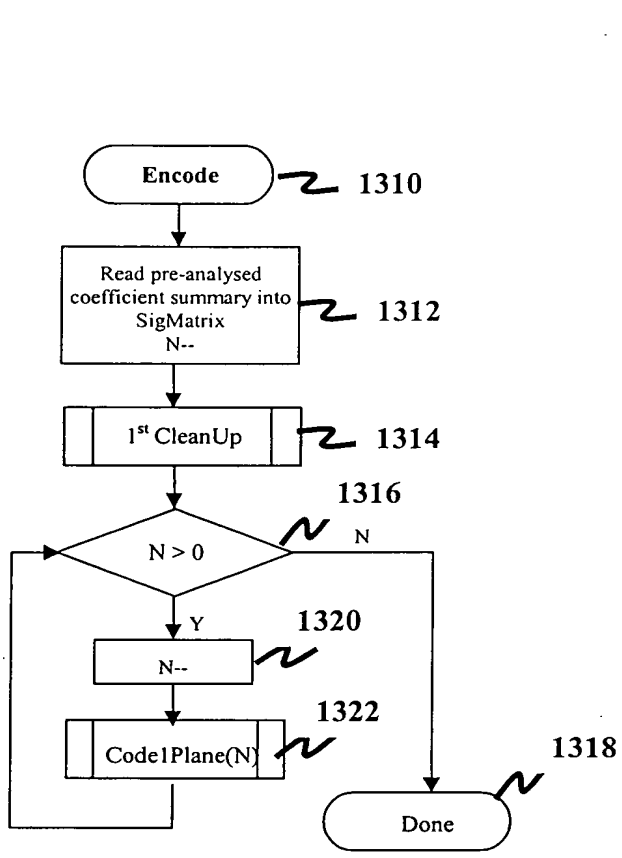
1300
Encode
1310
Read pre-analysed coefficient summary into SigMatrix
N--
1312
1st CleanUp
1314
1316
N > 0
N
Y
1320
N--
1322
Code1Plane(N)
1318
Done

FIG. 13

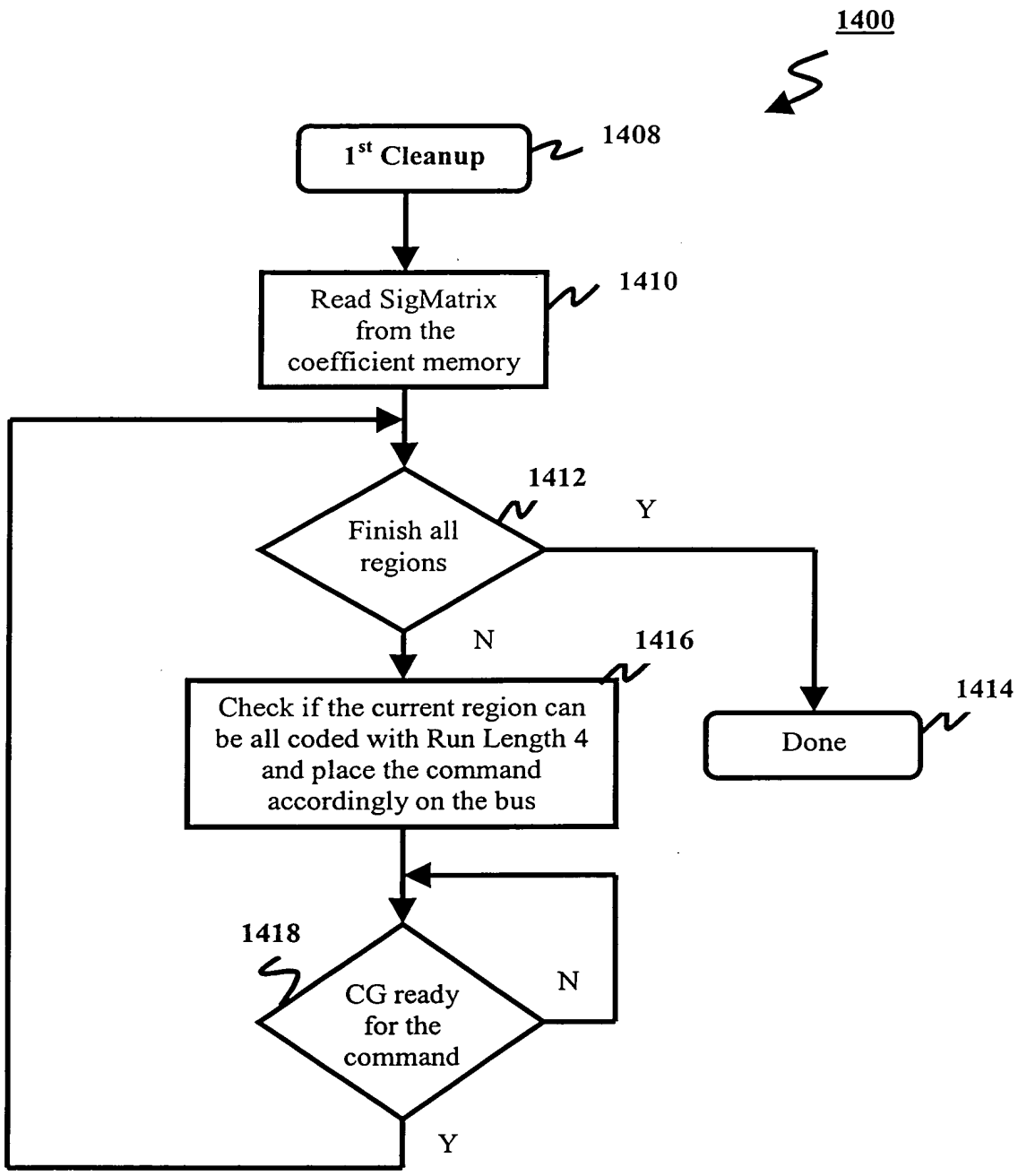
1400
1st Cleanup
1408
Read SigMatrix from the coefficient memory
1410
1412
Finish all regions
Y
N
1416
Check if the current region can be all coded with Run Length 4 and place the command accordingly on the bus
1414
Done
1418
CG ready for the command
N
Y

FIG. 14

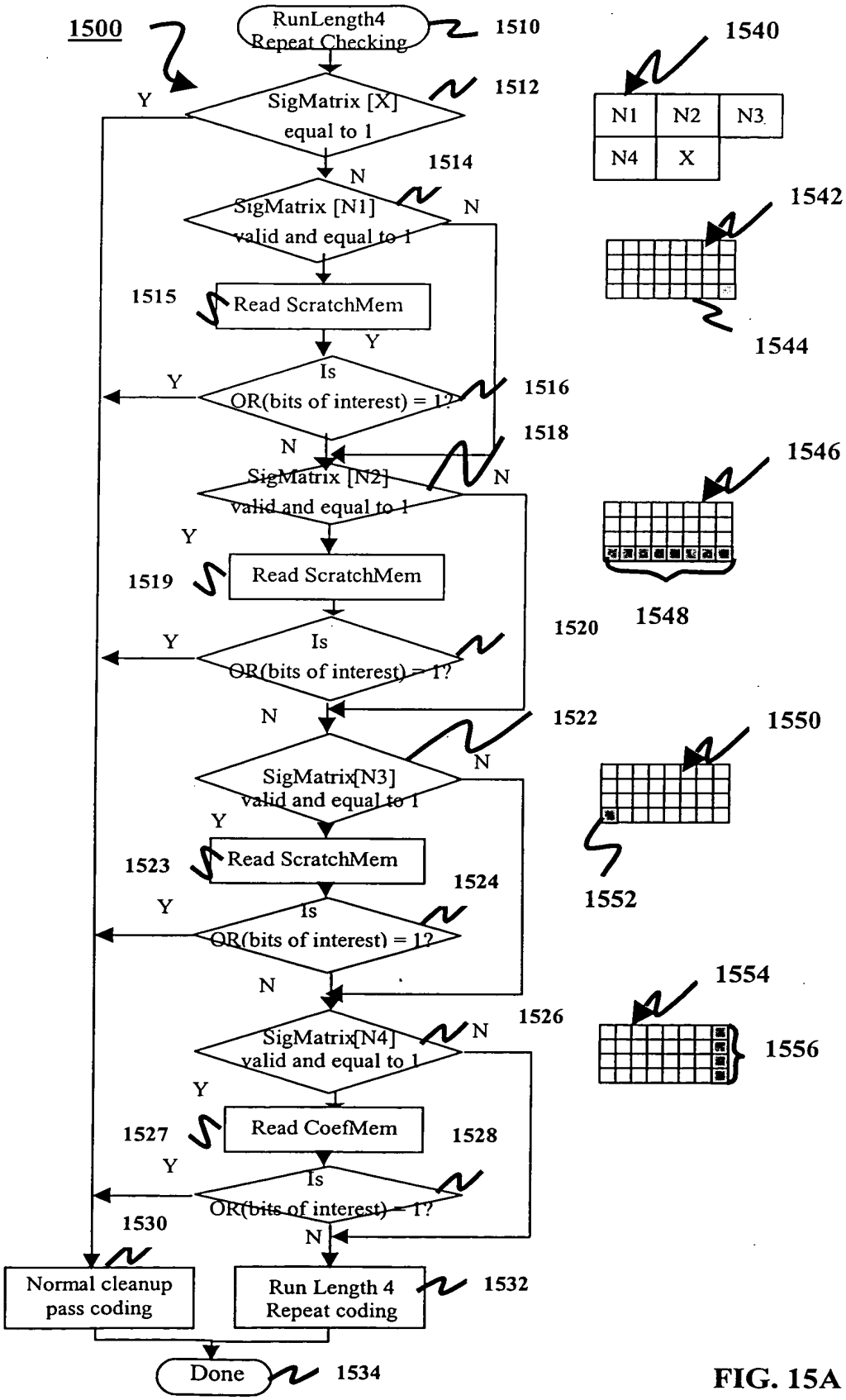
1500
RunLength4 Repeat Checking
1510
SigMatrix [X] equal to 1
1512
Y
N
1514
SigMatrix [N1] valid and equal to 1
N
1515
Read ScratchMem
Y
Is OR(bits of interest) = 1?
Y
1516
N
1518
SigMatrix [N2] valid and equal to 1
N
Y
1519
Read ScratchMem
Is OR(bits of interest) = 1?
Y
1520
N
1522
SigMatrix[N3] valid and equal to 1
N
Y
1523
Read ScratchMem
1524
Is OR(bits of interest) = 1?
Y
N
1526
SigMatrix[N4] valid and equal to 1
N
Y
1527
Read CoefMem
1528
Is OR(bits of interest) = 1?
Y
1530
N
Normal cleanup pass coding
Run Length 4 Repeat coding
1532
Done
1534
1540
N1
N2
N3
N4
X
1542
1544
1546
1548
1550
1552
1554
1556

FIG. 15A

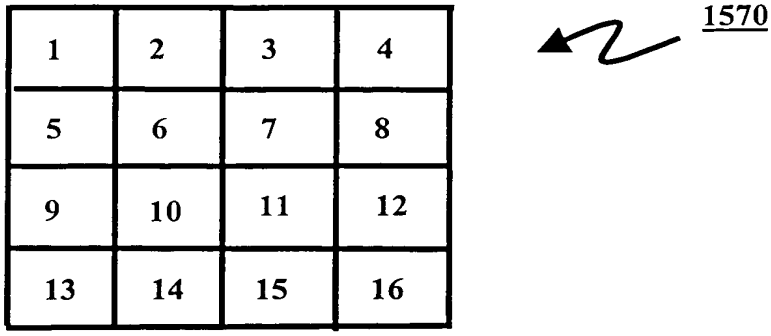

FIG. 15B

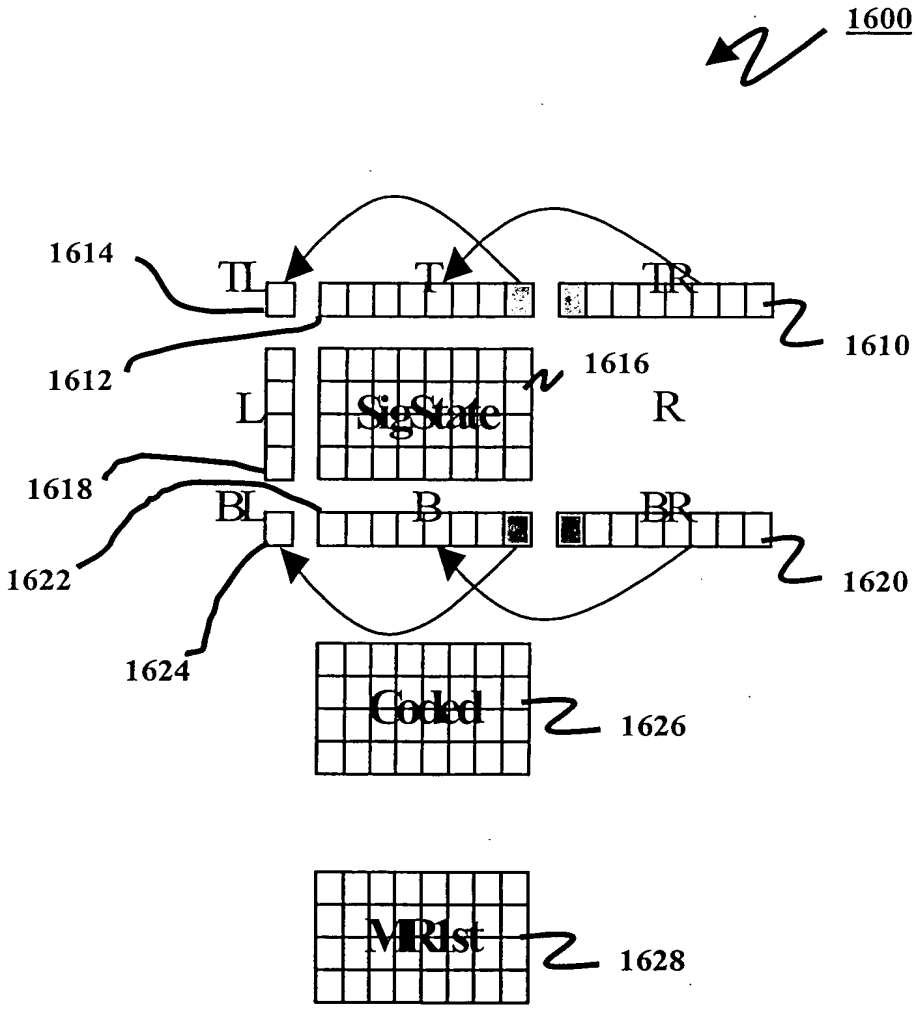
1600
1614
TL
T
TR
1610
1612
1616
L
SigState
R
1618
BL
B
BR
1622
1620
1624
Coded
1626
MRIst
1628

FIG. 16

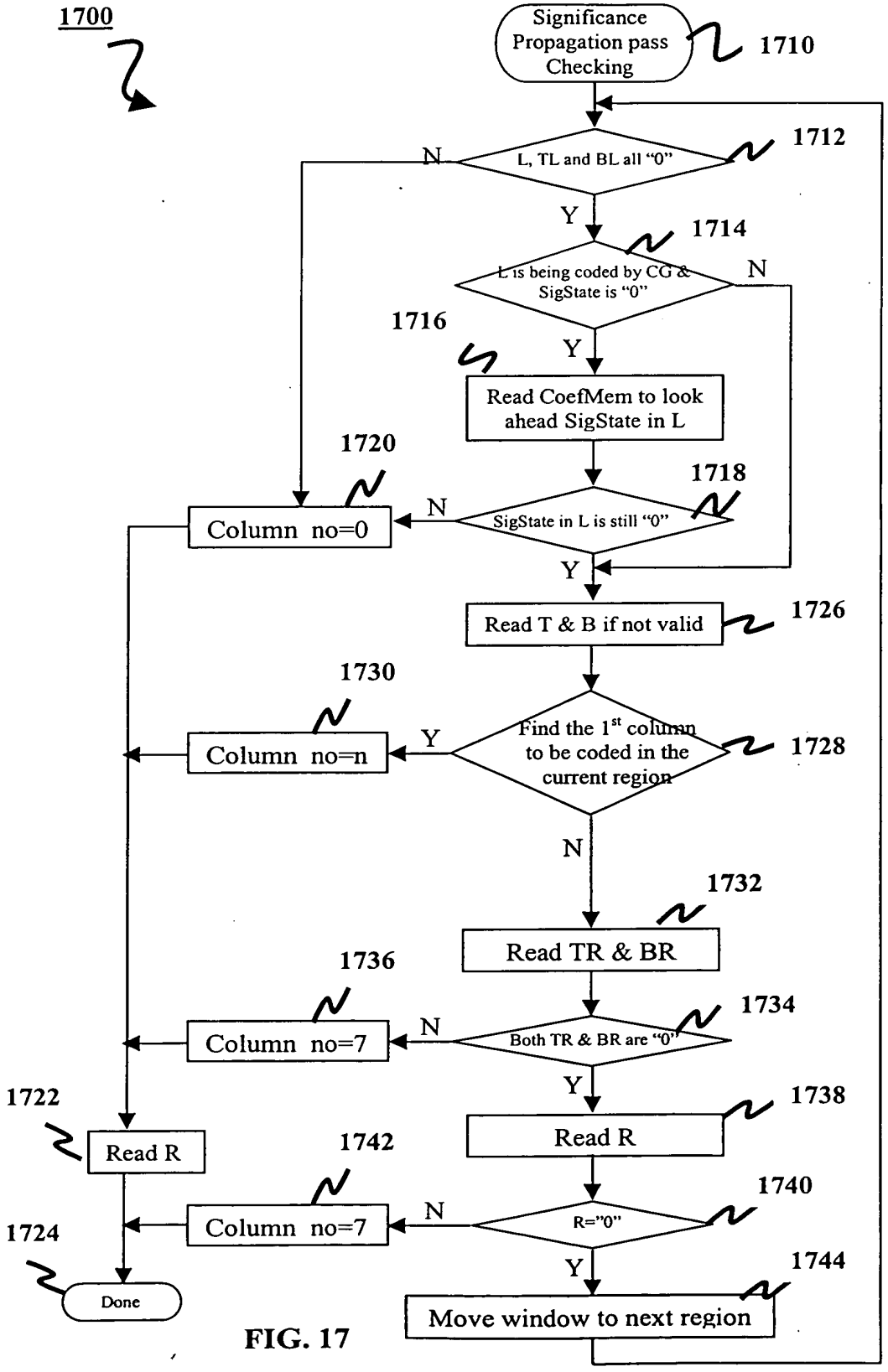
1700
Significance Propagation pass Checking
1710
L, TL and BL all "0"
1712
N
Y
L is being coded by CG & SigState is "0"
1714
N
Y
1716
Read CoefMem to look ahead SigState in L
SigState in L is still "0"
1718
N
Y
1720
Column no=0
Read T & B if not valid
1726
Find the 1st column to be coded in the current region
1728
Y
1730
Column no=n
N
Read TR & BR
1732
Both TR & BR are "0"
1734
N
1736
Column no=7
Y
Read R
1738
R="0"
1740
N
1742
Column no=7
Y
1744
Move window to next region
1722
Read R
1724
Done

FIG. 17

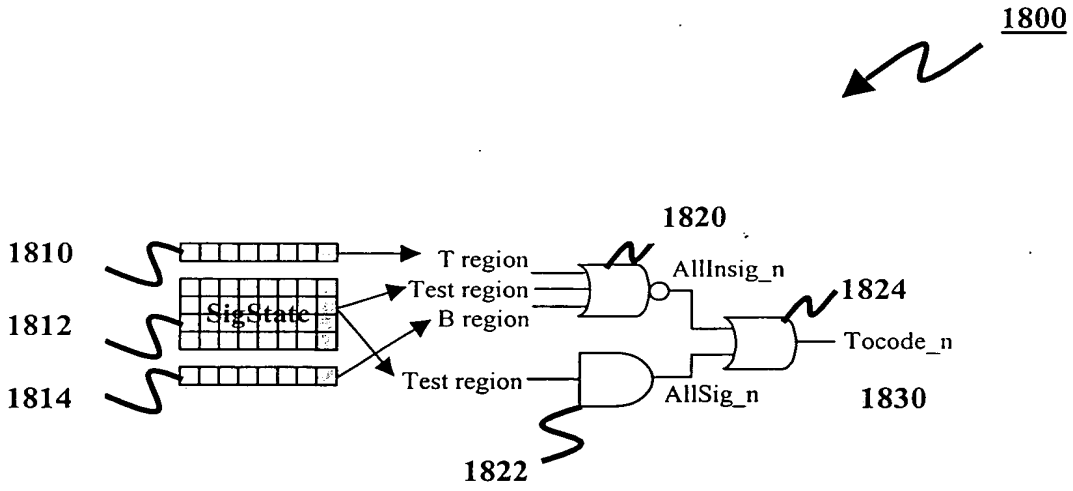
1800
1820
1810
T region
AllInsig_n
1824
1812
SigState
Test region
B region
Tocode_n
1814
Test region
AllSig_n
1830
1822

FIG. 18

1900

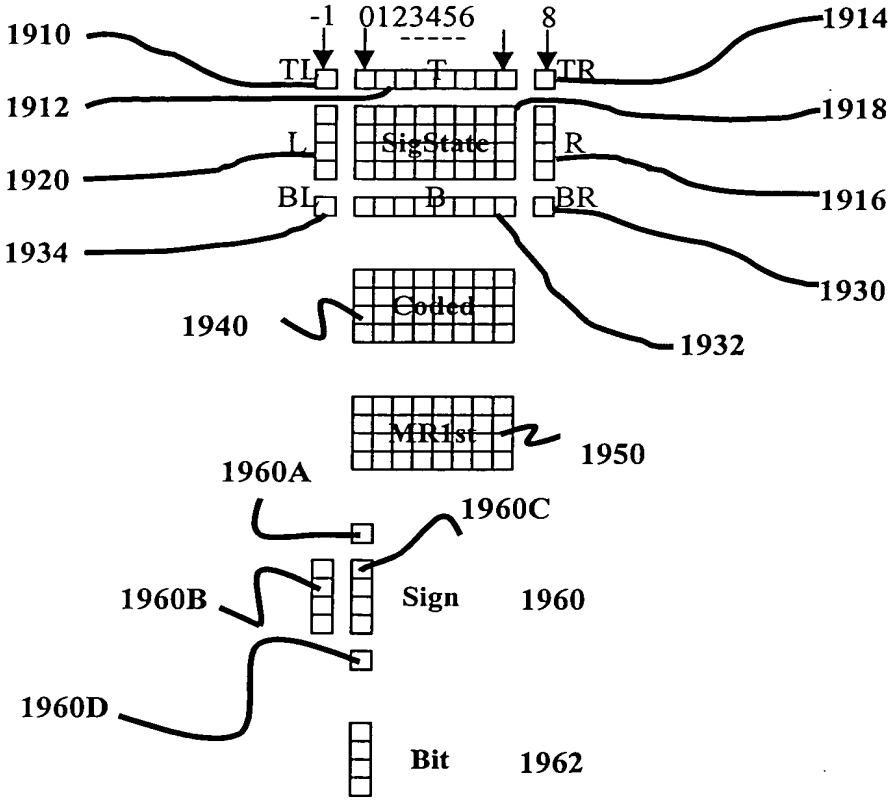
-1 0123456 8
1910
1914
TL
T
TR
1912
1918
L
SigState
R
1920
1916
BL
B
BR
1934
1930
1940
Coded
1932
MR1st
1950
1960A
1960C
1960B
Sign
1960
1960D
Bit
1962

FIG. 19

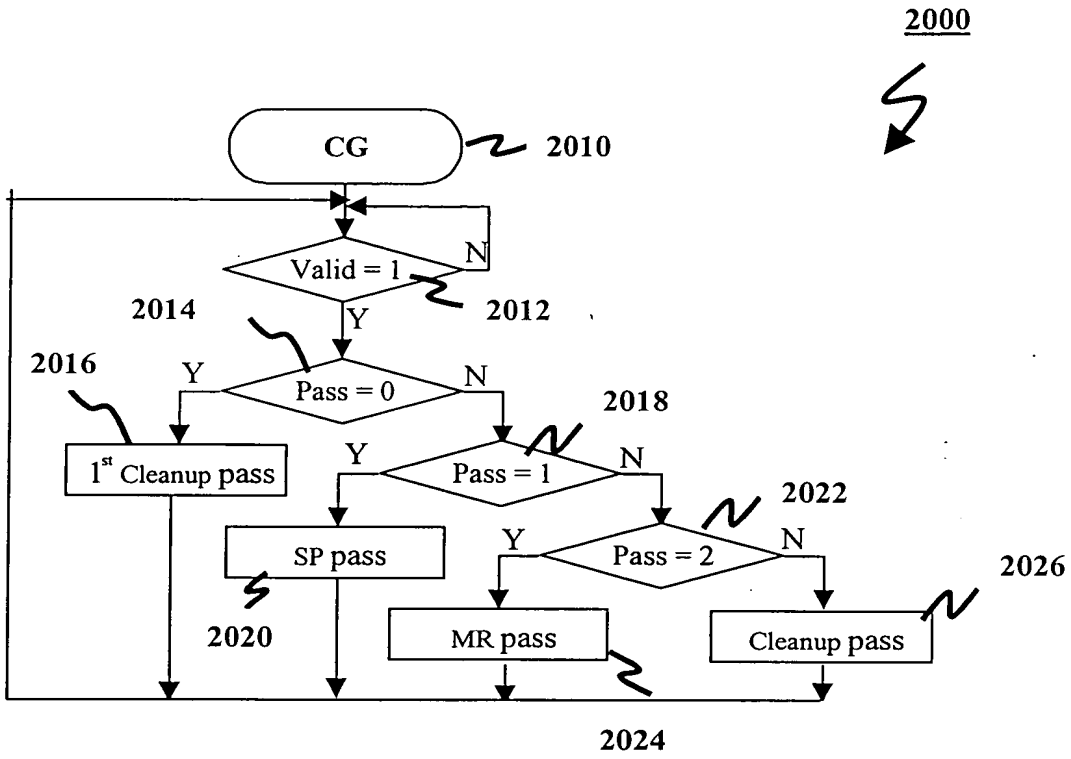
2000
CG
2010
Valid = 1
N
2012
Y
2014
Pass = 0
Y
N
2016
2018
1st Cleanup pass
Pass = 1
Y
N
2022
SP pass
Pass = 2
Y
N
2026
2020
MR pass
Cleanup pass
2024

FIG. 20

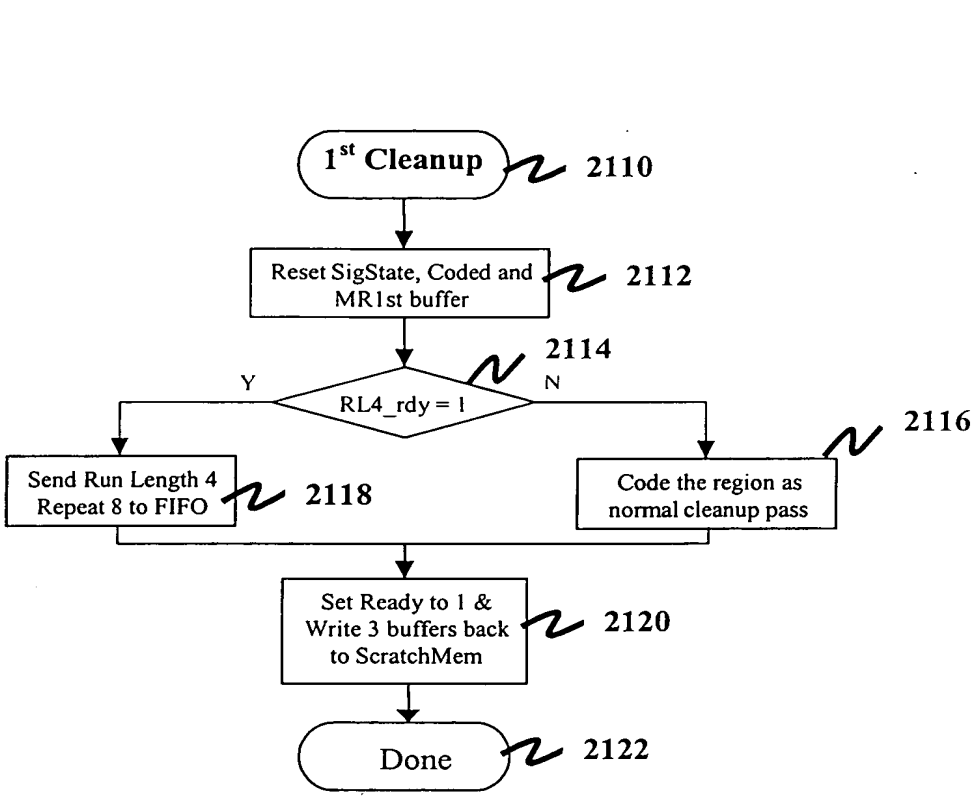
2100
1st Cleanup
2110
Reset SigState, Coded and MR1st buffer
2112
2114
Y
RL4_rdy = 1
N
2116
Send Run Length 4 Repeat 8 to FIFO
2118
Code the region as normal cleanup pass
Set Ready to 1 & Write 3 buffers back to ScratchMem
2120
Done
2122

FIG. 21

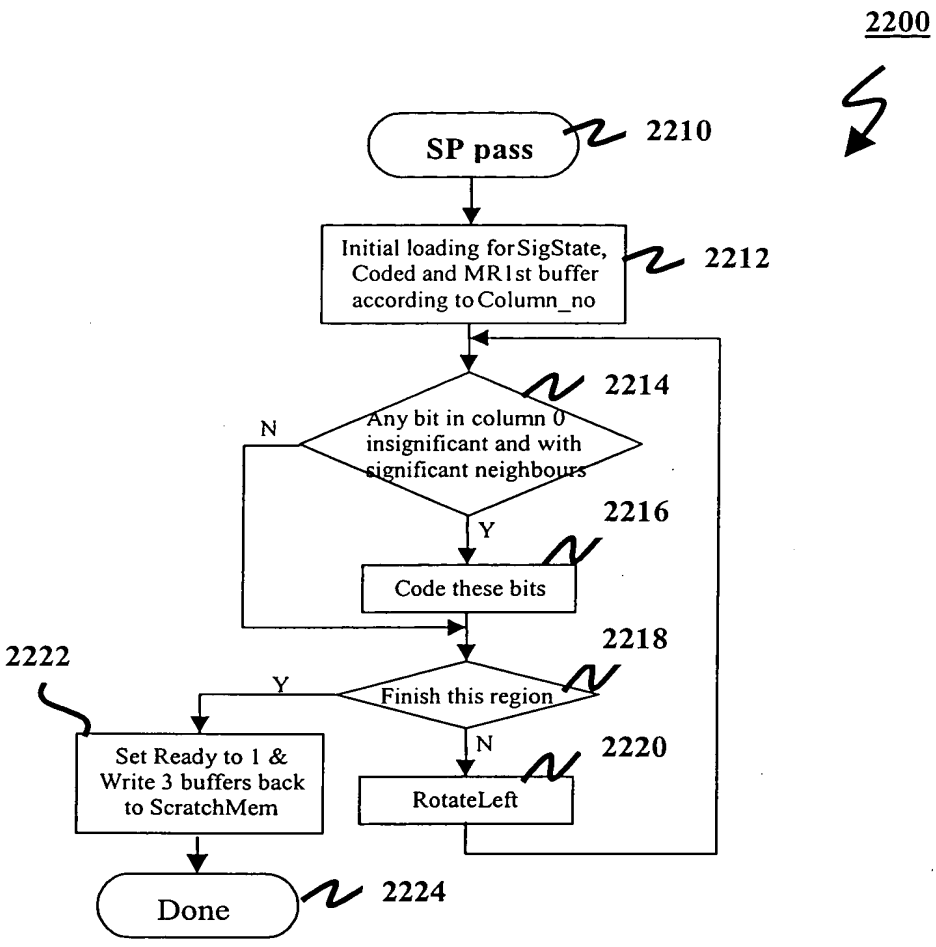
2200
SP pass
2210
Initial loading for SigState, Coded and MR1st buffer according to Column_no
2212
2214
Any bit in column 0 insignificant and with significant neighbours
N
Y
2216
Code these bits
2218
Finish this region
Y
N
2220
RotateLeft
2222
Set Ready to 1 & Write 3 buffers back to ScratchMem
Done
2224

FIG. 22

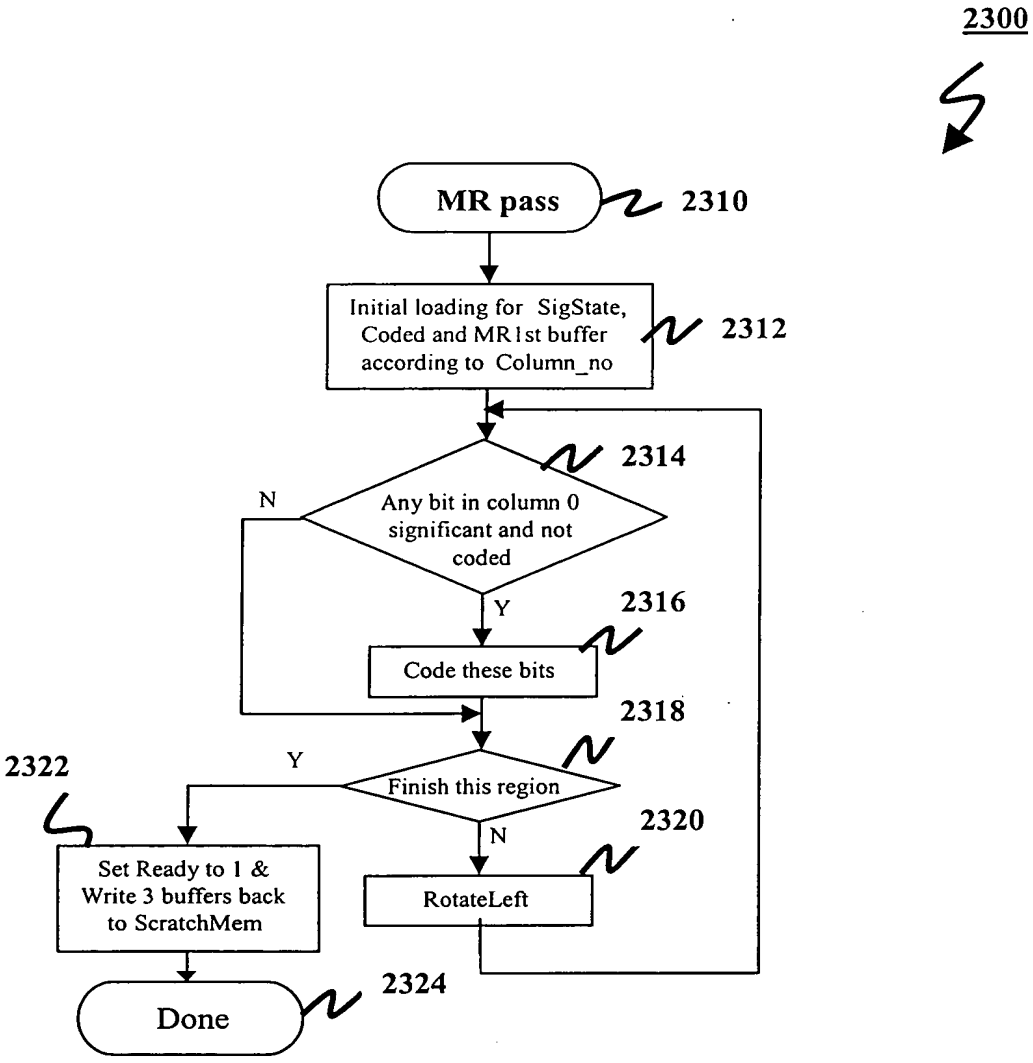
2300
MR pass
2310
Initial loading for SigState, Coded and MR1st buffer according to Column_no
2312
2314
Any bit in column 0 significant and not coded
N
Y
2316
Code these bits
2318
2322
Y
Finish this region
N
2320
Set Ready to 1 & Write 3 buffers back to ScratchMem
RotateLeft
Done
2324

FIG. 23

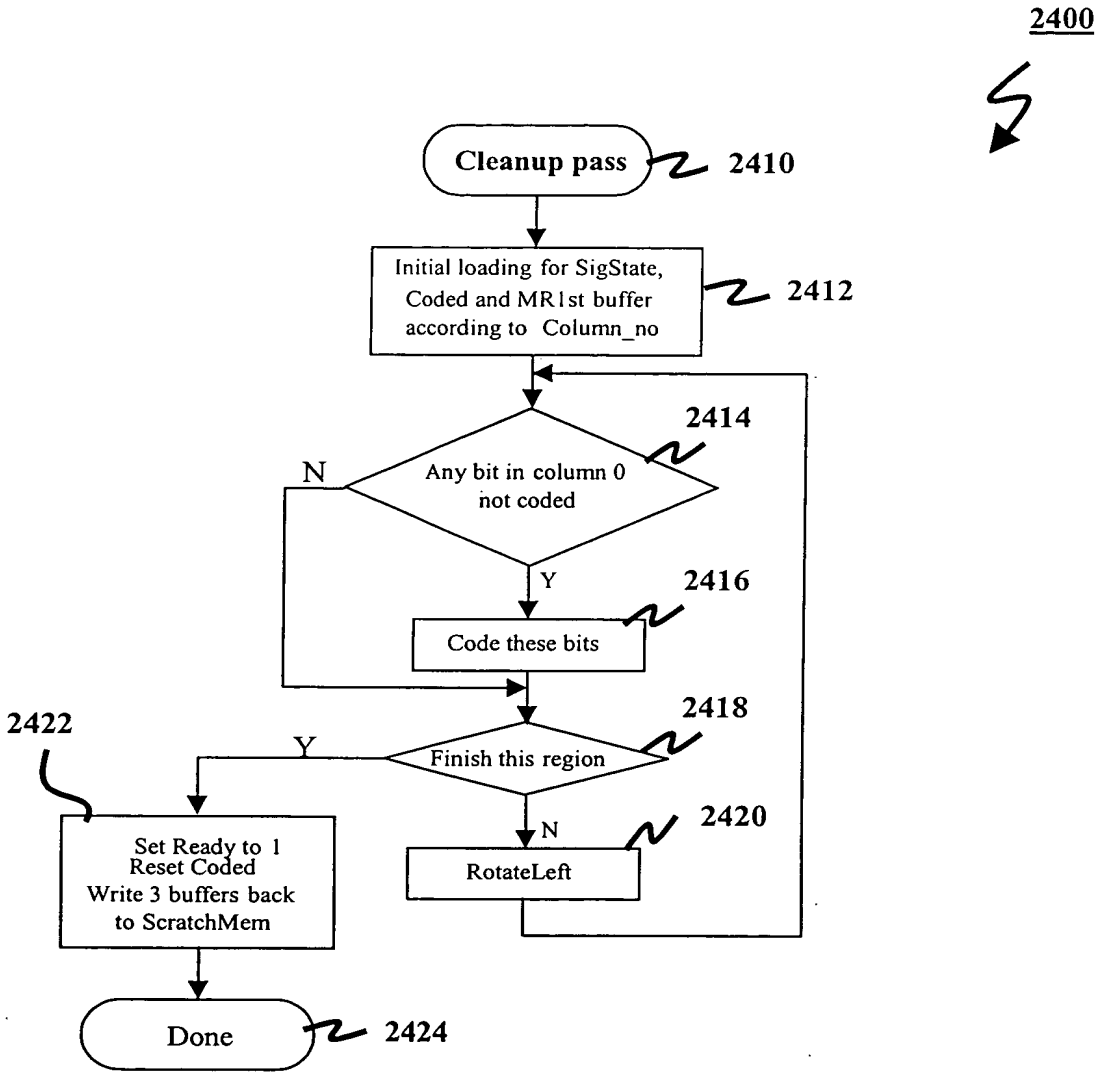
2400
Cleanup pass
2410
Initial loading for SigState, Coded and MR1st buffer according to Column_no
2412
2414
Any bit in column 0 not coded
N
Y
2416
Code these bits
2418
Finish this region
2422
Y
N
2420
Set Ready to 1
Reset Coded
Write 3 buffers back to ScratchMem
RotateLeft
Done
2424

FIG. 24

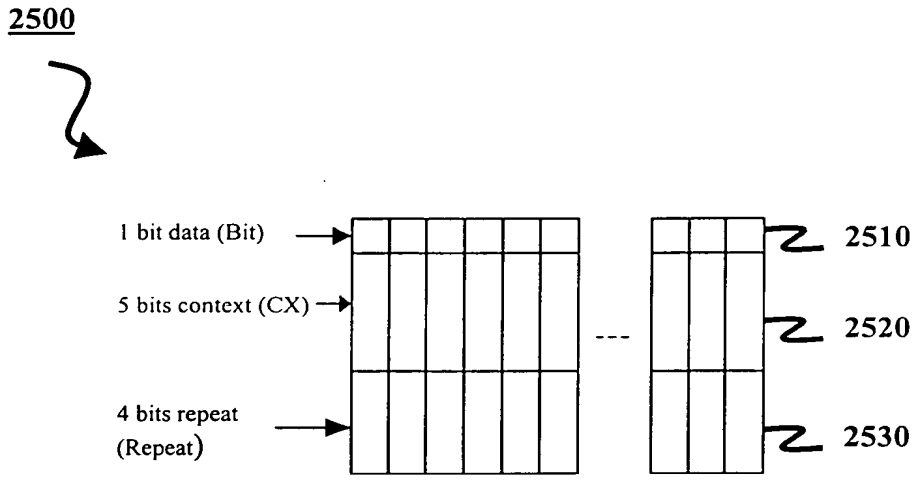
2500
1 bit data (Bit)
5 bits context (CX)
4 bits repeat
(Repeat)
2510
2520
2530

FIG. 25

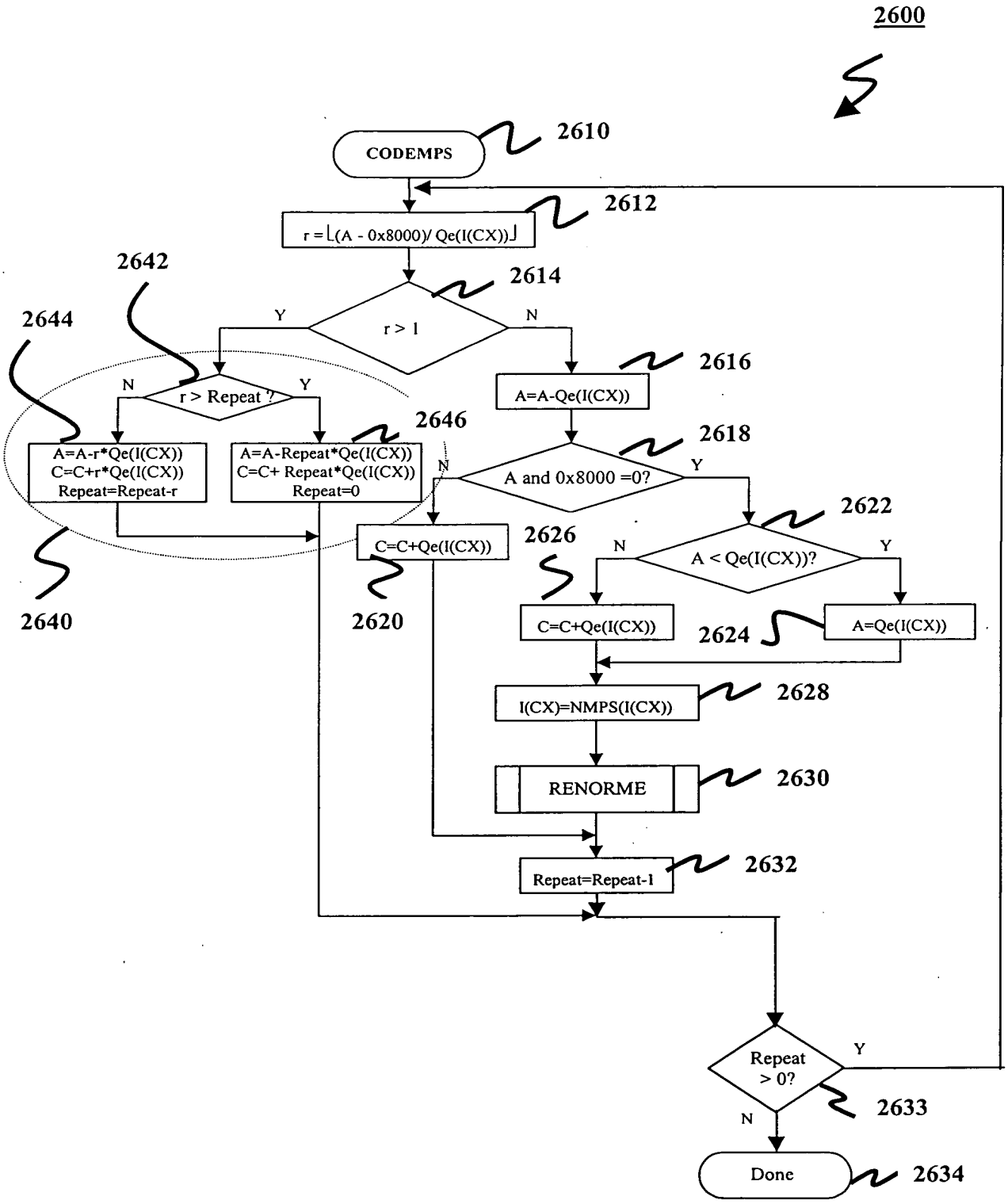
2600
CODEMPS
2610
r = ⌊(A - 0x8000)/ Qe(I(CX))⌋
2612
2642
2614
2644
Y
r > 1
N
2616
A=A-Qe(I(CX))
N
r > Repeat ?
Y
2646
2618
A=A-r*Qe(I(CX))
C=C+r*Qe(I(CX))
Repeat=Repeat-r
A=A-Repeat*Qe(I(CX))
C=C+ Repeat*Qe(I(CX))
Repeat=0
N
A and 0x8000 =0?
Y
2622
2626
C=C+Qe(I(CX))
N
A < Qe(I(CX))?
Y
2640
2620
C=C+Qe(I(CX))
2624
A=Qe(I(CX))
I(CX)=NMPS(I(CX))
2628
RENORME
2630
Repeat=Repeat-1
2632
Repeat
> 0?
Y
2633
N
Done
2634

FIG. 26

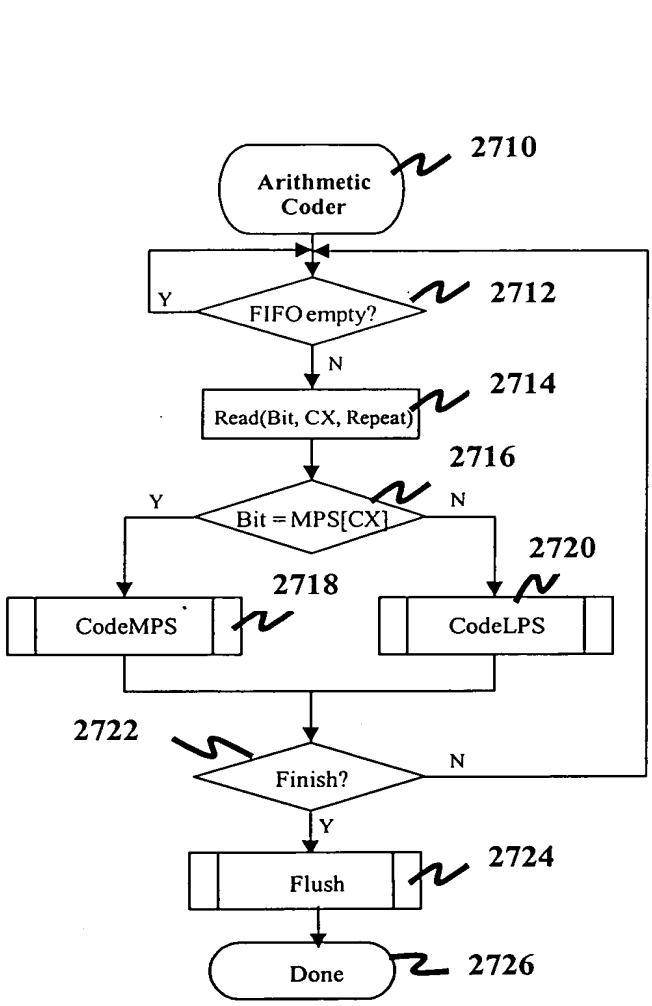
2700
2710
Arithmetic Coder
2712
Y
FIFO empty?
N
2714
Read(Bit, CX, Repeat)
2716
Y
Bit = MPS[CX]
N
2720
2718
CodeMPS
CodeLPS
2722
Finish?
N
Y
2724
Flush
Done
2726

FIG. 27

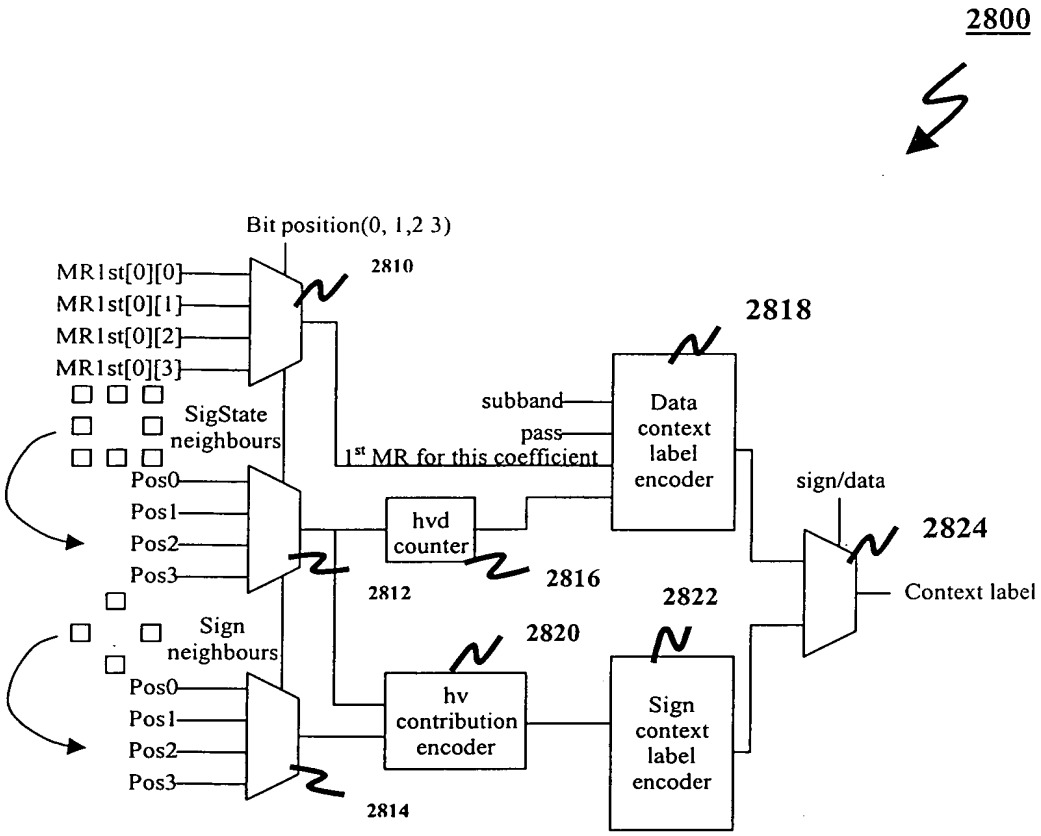
2800
Bit position(0, 1,2 3)
2810
MR1st[0][0]
MR1st[0][1]
MR1st[0][2]
MR1st[0][3]
SigState neighbours
Pos0
Pos1
Pos2
Pos3
2812
Sign neighbours
Pos0
Pos1
Pos2
Pos3
2814
subband
pass
1st MR for this coefficient
2818
Data context label encoder
hvd counter
2816
2820
hv contribution encoder
2822
Sign context label encoder
sign/data
2824
Context label

FIG. 28